HZ BOOKS
华章科技

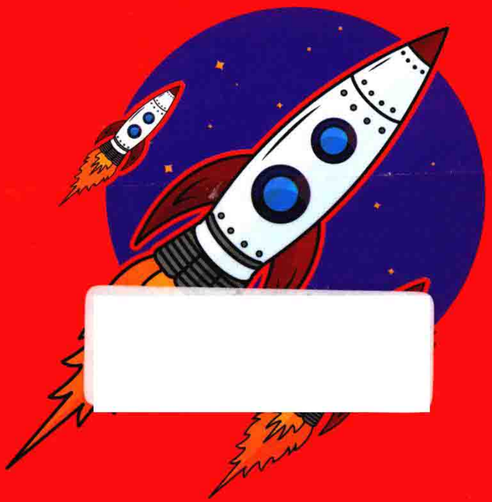
SpringBoot揭秘

快速构建微服务体系

王福强 著

阿里与平安集团技术高层倾心倾情推荐，互联网与互联网金融行业各大技术掌门一致好评。

理论与实践相结合、框架与生态相结合、技术与产品相结合，多视角、多维度、多场景地为大家深刻揭示了SpringBoot微服务框架和微服务架构体系的终极奥秘。



SpringBoot Unleashed

Microservices Quick Starter

机械工业出版社
China Machine Press

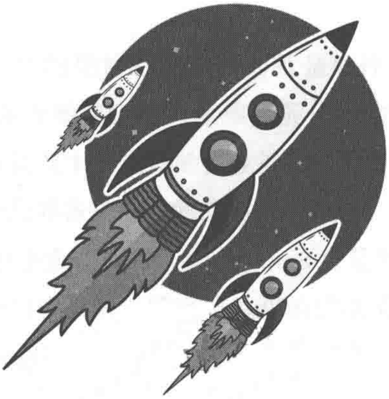
VISIT…

LANZAROTE Caliente.com

SpringBoot揭秘

快速构建微服务体系

王福强 著



SpringBoot Unleashed

Microservices Quick Starter

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

SpringBoot 揭秘：快速构建微服务体系 / 王福强著．—北京：机械工业出版社，2016.5

ISBN 978-7-111-53664-2

Ⅰ. S…　Ⅱ. 王…　Ⅲ. 互联网络－网络服务器　Ⅳ. TP368.5

中国版本图书馆 CIP 数据核字（2016）第 091203 号

SpringBoot 揭秘：快速构建微服务体系

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）

责任编辑：李　艺　　责任校对：殷　虹

印　　刷：北京瑞德印刷有限公司　　版　　次：2016 年 5 月第 1 版第 1 次印刷

开　　本：170mm × 242mm　1/16　　印　　张：12.5

书　　号：ISBN 978-7-111-53664-2　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426　88361066　　投稿热线：（010）88379604

购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

Foreword 1 推荐序 1

2015 年技术圈最火的名词大概就是微服务了。国内外的互联网技术会议上，但凡分享题目中包含“MicroService”，不论内容质量如何，一定人山人海、摩肩接踵。

追本溯源，服务化的架构思想十年前就是软件架构的标准范式。淘宝和阿里在 2007 年左右就开始奠定了大规模服务化架构的基础，经过几代架构师的努力，有了今天承载双十一规模的商业操作系统。这中间诞生的很多优秀的 Java 中间件也成为开源界备受追崇的范例。

但是对于很多中小企业而言，SpringBoot 会是另一个性价比极高的选择。福强的这本书出现得恰逢其时，既有体系化的理论又不乏有价值的实践。对于想了解微服务和 SpringBoot 的架构师而言，是难得的修炼秘籍。

南天（本名是庄卓然） 阿里巴巴资深总监

推荐序 2 *Foreword 2*

多年前，第一次见福强，就知道他在写书，那时就是关于 Spring 的书籍。等到出书后，我翻看之下，发现福强写得非常实用。

时隔若干年，福强又来信告知有新作问世，这是他经历几年的大型网站实践之后，在创业阶段写的书。在这个阶段还能坚持写作的人非常少，足以说明他对技术的执着和坚持。有了成熟大型网站和创业阶段的实践经验，本书不仅是 SpringBoot 的指南，还是各种实战经验的提炼和总结。福强不仅在 Java，在 Scala、Golang 方面都有颇深的理解，这种跨语言方面对技术的融会贯通也为整个构建过程起着催化剂的作用。福强这次给大家带来的这本书，从不同角度对微服务这一热门话题进行了介绍和探讨，同时加入了自己多年的实践经验，值得一读。

Eric（中文名是王齐） 平安好医生 CTO

Preface 序　言

随着微服务（Micro Service）理念的盛行，一个流行的概念也随之诞生——微框架（Micro Framework），而其中最耀眼的，当属 SpringBoot。

虽然 Dropwizard 是公认的最早的微框架，但 SpringBoot“青出于蓝而胜于蓝”，背靠 Spring 框架衍生出来的整个生态体系，无论是从“出身”，还是社区的支撑上，SpringBoot 都是微框架选型的不二之选。

实际上，SpringBoot 并非单单一个微框架的概念就可以概括，笔者认为将 SpringBoot 看作一种最佳实践会更为贴切：一种 Spring 框架及其社区对“约定优先于配置”（Convention Over Configuration）理念的最佳实践。

温故而知新，笔者将通过本书带领大家回顾 Spring 框架的历史，进而引领大家探索 SpringBoot 框架的来龙去脉，最终引领大家去探索基于 SpringBoot 的微服务实践之路。希望各位能够享受这段文字旅程并有所收获。

前　言 Preface

为什么写这本书

忘了是2015年的哪一天，只记得几个朋友跟友商的其他几个做技术的朋友吃饭，并简单做下技术交流。席间，友商的几位朋友对SpringBoot框架实施微服务很感兴趣，交谈甚欢之际，我无意间开玩笑说："是不是该考虑写一本SpringBoot的书？"钟伦甫（原淘宝聚石）同学随口一句，"你倒是写啊！"，得，以行践言吧，谁让你把话说出去了呢？

当然，朋友的"热切期盼"只是其一，微服务盛行也是本书写作的一个契机，希望本书成为国内第一本微服务相关的原创图书，借此跟大家分享我对微服务的浅薄理解，并围绕SpringBoot微框架打造一套微服务体系可能的探索方向，权作抛砖引玉。如果不同的思想可以借此激荡和碰撞形成更多共鸣，则吾之幸甚。

因工作繁忙，只能抽取零碎时间躬耕于晨曦和月光之下，经点滴积累，才终成此书，希望大家阅读愉快。

本书的主要内容和特色

本书以介绍微服务的基本概念开篇，逐步引出Java平台下打造微服务的利器——SpringBoot微框架。书中从SpringBoot微框架的"出身"开始，循序渐进，一步步为大家剖析SpringBoot微框架的设计理念和原理，并对框架的重点

功能和模块进行了逐一讲解。

当然，这还只是“前戏”，本书最精彩的部分在于，在大家对 SpringBoot 微框架已经有了基本的认识之后，我们将一起探索如何基于 SpringBoot 微框架打造一套完备的微服务体系。因为如果没有平台化体系化的基础支撑，空谈微服务将无太大意义。

SpringBoot 微框架依托 Java 平台和 Spring 框架，拥有良好的可扩展性和可定制性，为了说明这一点，我们单独开辟了一章内容，为大家介绍如何使用 Scala 和 SpringBoot 微框架来开发和交付相应的微服务，并且围绕 Scala 和 SpringBoot 如何打造相应的工具，技术产品等支持来提高相应微服务的交付效率。

最后我会与大家一起对 SpringBoot 微框架的相关内容进行回顾和展望，以期温故而知新。

本书总体上可以总结为三个关键词，“框架、体系、生态”，三者循序渐进，相辅相成，在使用 SpringBoot 微框架打造自己特色的微服务体系和技术生态之时，希望大家记住这三个关键词。

本书面向的读者

本书希望面向的读者当然是那些对 SpringBoot 微框架感兴趣的同学，如果你想了解 SpringBoot 微框架，并且尝试进一步深入定制该框架以满足自己团队和公司的需要，也希望会对你有所启发。

除此之外还包括：

- Java 平台上的广大研发同学，可以借此书了解业界微服务相关的最新动态。
- 其他平台上的广大研发同学，可借此书“管中窥豹”，了解微服务的一般体系和生态建设，对比并引入自身的技术和微服务体系建设之中。
- 脱离技术一线已久的技术负责人。

如何阅读本书

本书采用循序渐进的形式编写，所以顺序阅读是推荐的阅读方式。

勘误和资源

鉴于一家之言且编撰仓促，难免会有所纰漏，观点有失偏颇，所以，我在github 网站上专门新建了一个 issue 项目（https://github.com/fujohnwang/unveil-springboot-feedbacks），如果大家在阅读此书之后发现有哪些错误和疑问，或者改进建议，可以在此项目上新建 issue 来表达自己的观点和建议。如果时间不充裕，我会适时地选择性给予答复，当然，更希望大家可以通过 issue 展开讨论，互相切磋和解答疑问 。

致谢

除了最初的一句戏言，钟伦甫同学也是本书的第一位读者，帮助审稿并提出很多建议，所以，本书得以出版，第一需要感谢的就是钟伦甫同学。

其次，我要感谢华章出版社的杨福川和李艺，福川兄在接到我的出版意向之后，快速地跟进和落实，在本书初稿编写完成时马上着手出版，诸位得以在2016 年上半年就手捧此书，皆需感谢福川兄的重点关注和推进。

最后要感谢我的父母，感谢他们把我带到这个世界上并让我做自己想做和要做的事情。

Contents 目　录

推荐序 1

推荐序 2

序言

前言

第 1 章　了解微服务 ········· 1

1.1　什么是微服务 ········· 1

1.2　微服务因何而生 ········· 2

1.3　微服务会带来哪些好处 ········· 4

1.3.1　独立，独立，还是独立 ········· 4

1.3.2　多语言生态 ········· 6

1.4　微服务会带来哪些挑战 ········· 8

1.5　本章小结 ········· 9

第 2 章　饮水思源：回顾与探索 Spring 框架的本质 ········· 11

2.1　Spring 框架的起源 ········· 11

2.2　Spring IoC 其实很简单 ········· 12

2.3　了解一点儿 JavaConfig ········· 14

2.3.1 那些高曝光率的 Annotation …… 17
2.4 本章小结 …… 18

第 3 章 SpringBoot 的工作机制 …… 19
3.1 SpringBoot 初体验 …… 19
3.2 @SpringBootApplication 背后的秘密 …… 20
3.2.1 @Configuration 创世纪 …… 21
3.2.2 @EnableAutoConfiguration 的功效 …… 22
3.2.3 可有可无的 @ComponentScan …… 25
3.3 SpringApplication：SpringBoot 程序启动的一站式解决方案 …… 26
3.3.1 深入探索 SpringApplication 执行流程 …… 27
3.3.2 SpringApplicationRunListener …… 30
3.3.3 ApplicationListener …… 31
3.3.4 ApplicationContextInitializer …… 32
3.3.5 CommandLineRunner …… 33
3.4 再谈自动配置 …… 34
3.4.1 基于条件的自动配置 …… 34
3.4.2 调整自动配置的顺序 …… 35
3.5 本章小结 …… 35

第 4 章 了解纷杂的 spring-boot-starter …… 37
4.1 应用日志和 spring-boot-starter-logging …… 39
4.2 快速 Web 应用开发与 spring-boot-starter-web …… 40
4.2.1 项目结构层面的约定 …… 41
4.2.2 SpringMVC 框架层面的约定和定制 …… 41
4.2.3 嵌入式 Web 容器层面的约定和定制 …… 42
4.3 数据访问与 spring-boot-starter-jdbc …… 43
4.3.1 SpringBoot 应用的数据库版本化管理 …… 46

4.4 spring-boot-starter-aop 及其使用场景说明 …… 48
4.4.1 spring-boot-starter-aop 在构建 spring-boot-starter-metrics 自定义模块中的应用 …… 49
4.5 应用安全与 spring-boot-starter-security …… 58
4.5.1 了解 SpringSecurity 基本设计 …… 61
4.5.2 进一步定制 spring-boot-starter-security …… 66
4.6 应用监控与 spring-boot-starter-actuator …… 68
4.6.1 自定义应用的健康状态检查 …… 70
4.6.2 开放的 endpoints 才真正“有用” …… 73
4.6.3 用还是不用，这是个问题 …… 75
4.7 本章小结 …… 77

第 5 章 SpringBoot 微服务实践探索 …… 79

5.1 使用 SpringBoot 构建微服务 …… 79
5.1.1 创建基于 Dubbo 框架的 SpringBoot 微服务 …… 80
5.1.2 使用 SpringBoot 快速构建 Web API …… 91
5.1.3 使用 SpringBoot 构建其他形式的微服务 …… 104
5.2 SpringBoot 微服务的发布与部署 …… 110
5.2.1 spring-boot-starter 的发布与部署方式 …… 112
5.2.2 基于 RPM 的发布与部署方式 …… 115
5.2.3 基于 Docker 的发布与部署方式 …… 120
5.3 SpringBoot 微服务的注册与发现 …… 124
5.4 SpringBoot 微服务的监控与运维 …… 127
5.4.1 推还是拉，这一直是个问题 …… 131
5.4.2 从局部性触发式报警到系统性智能化报警 …… 132
5.5 SpringBoot 微服务的安全与防护 …… 133
5.6 SpringBoot 微服务体系的脊梁：发布与部署平台 …… 135
5.7 本章小结 …… 138

第 6 章　SpringBoot 与 Scala ······ 139

6.1　使用 Maven 构建和发布基于 SpringBoot 的 Scala 应用 ······ 140

6.1.1　进一步简化基于 Maven 的 Scala 项目创建 ······ 146

6.1.2　进一步简化基于 Scala 的 Web API 开发 ······ 167

6.2　使用 SBT 构建和发布基于 SpringBoot 的 Scala 应用 ······ 174

6.2.1　探索基于 SBT 的 SpringBoot 应用开发模式 ······ 175

6.2.2　探索基于 SBT 的 SpringBoot 应用发布策略 ······ 181

6.3　本章小结 ······ 184

第 7 章　SpringBoot 总结与展望 ······ 186

第 1 章 Chapter 1

了解微服务

SpringBoot 是一个可使用 Java 构建微服务的微框架，所以在了解 SpringBoot 之前，我们需要先了解什么是微服务。

1.1 什么是微服务

微服务（Microservice）虽然是当下刚兴起的比较流行的新名词，但本质上来说，微服务并非什么新的概念。实际上，很多 SOA 实施成熟度比较好的企业，已经在使用和实施微服务了。只不过，它们只是在闷声发大财，并不介意是否有一个比较时髦的名词来明确表述 SOA 的这个发展演化趋势罢了。

微服务其实就是服务化思路的一种最佳实践方向，遵循 SOA 的思路，各个企业在服务化治理的道路上走的时间长了，踩的坑多了，整个软件交付链路上各个环节的基础设施逐渐成熟了，微服务自然而然就诞生了。

当然，之所以叫微服务，是与之前的服务化思路和实践相比较而来的。早些年的服务实现和实施思路是将很多功能从开发到交付都打包成一个很大的服务单元（一般称为 Monolith），而微服务实现和实施思路则更强调功能趋向单一，服务单元小型化和微型化。如果用“茶壶煮饺子”来打比方的话，原来我们是在一个茶壶里煮很多个饺子，现在（微服务化之后）则基本上是在一个茶

壶煮一个饺子，而这些饺子就是服务的功能，茶壶则是将这些服务功能打包交付的服务单元，如图 1-1 所示。

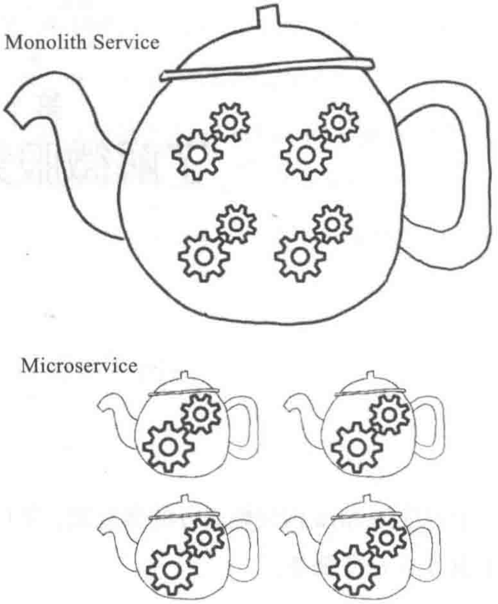


图 1-1　论茶壶里煮“饺子”的不同形式

所以，从思路和理念上来讲，微服务就是要倡导大家尽量将功能进行拆分，将服务粒度做小，使之可以独立承担对外服务的职责，沿着这个思路开发和交付的软件服务实体就叫作“微服务”，而围绕着这个思路和理念构建的一系列基础设施和指导思想，笔者将它称为“微服务体系”。

1.2　微服务因何而生

微服务的概念我们应该大体了解了，那么微服务又是怎么来的？原来将很多功能打包为一个很大的服务单元进行交付的做法不能满足需求吗？

实际上，并非原来“大一统”（Monolith）的服务化实践不能满足要求，也不是不好，只是，它有自己存在的合理场景。对于 Monolith 服务来说，如果团队不大，软件复杂度不高，那么，使用 Monolith 的形式进行服务化治理是比较合适的，而且，这种方式对运维和各种基础设施的要求也不高。

但是，随着软件系统的复杂度持续飙升，软件交付的效率要求更高，投入的人力以及各项资源越来越多，基于 Monolith 的服务化思路就开始“捉襟见肘”。

在开发阶段，如果我们遵循 Monolith 的服务化理念，通常会将所有功能的实现都统一归到一个开发项目下，但随着功能的膨胀，这些功能一定会分发给不同的研发人员进行开发，造成的后果就是，大家在提交代码的时候频繁冲突并需要解决这些冲突，单一的开发项目成为了开发期间所有人的工作瓶颈。

为了减轻这种苦恼，我们自然会将项目按照要开发的功能拆分为不同的项目，从而负责不同功能的研发人员就可以在自己的代码项目上进行开发，从而解决了大家无法在开发阶段并行开发的苦恼。

到了软件交付阶段，如果我们遵循 Monolith 的服务化理念，那么，我们一定是将所有这些开发阶段并行开发的项目集合到一起进行交付，这就涉及服务化早期实践中比较有名的“火车模型”，即交付的服务就像一辆火车，而这个服务相关的所有功能对应的项目成果，就是要装上火车车厢的一件件货物，交付的列车只有等到所有项目都开发测试完成后才可以装车出发，完成整个服务的交付。很显然，只要有一个车厢没有准备好货物（即功能项目未开发测试完成），火车就不能发车，服务就不能交付，这大大降低了服务的交付效率。如果每个功能项目可以各自独立交付，那么就不需要都等同一辆火车，各自出发就可以了。顺着这个思路，自然而然地，大家逐渐各自独立，每一个功能或者少数相近的功能作为单一项目开发完成后将作为一个独立的服务单元进行交付，从而在服务交付阶段，大家也能够并行不悖，各自演化而不受影响。

所以，随着服务和系统的复杂度逐渐飙升，为了能够在整个软件的交付链路上高效扩展，将独立的功能和服务单元进行拆分，从而形成一个一个的微服务是自然而然发生的事情。这就像打不同的战役一样，在双方兵力不多、战场复杂度不高的情况下，Monolith 的统一指挥调度方式是合适的；而一旦要打大的战役（类似于系统复杂度提升），双方一定会投入大量的兵力（软件研发团队的规模增长），如果还是在狭小甚至固定的战场上进行厮杀，显然施展不开！所以，小战役有小战役的打法，大战役有大战役的战法，而微服务实际上就是一种帮助扩展组织能力、提升团队效率的应对“大战役”的方法，它帮助我们从软件开发到交付，进而到团队和组织层面多方位进行扩展。

总的来说，一方面微服务可以帮助我们应对飙升的系统复杂度；另一个方面，微服务可以帮助我们进行更大范围的扩展，从开发阶段项目并行开发的扩展，到交付阶段并行交付的扩展，再到相应的组织结构和组织能力的扩展，皆因微服务而受惠。

1.3 微服务会带来哪些好处

显然，随着系统复杂度的提升，以及对系统扩展性的要求越来越高，微服务化是一个很好的方向，但除此之外，微服务还会给我们带来哪些好处？

1.3.1 独立，独立，还是独立

我们说微服务打响的是各自的独立战争，所以，每一个微服务都是一个小王国，这些微服务跳出了“大一统”（Monolith）王国的统治，开始从各个层面打造自己的独立能力，从而保障自己的小王国可以持续稳固的运转。

首先，在开发层面，每个微服务基本上都是各自独立的项目（project），而对应各自独立项目的研发团队基本上也是独立对应，这样的结构保证了微服务的并行研发，并且各自快速迭代，不会因为所有研发都投入一个近乎单点的项目，从而造成开发阶段的瓶颈。开发阶段的独立，保证了微服务的研发可以高效进行。

服务开发期间的形态，跟服务交付期间的形态原则上是不需要完全高度统一的，即使我们在开发的时候都是各自进行，但交付的时候还是可以一起交付，不过这不是微服务的做法。在微服务治理体系下，各个微服务交付期间也是各自独立交付的，从而使得每个微服务从开发到交付整条链路上都是独立进行，这大大加快了微服务的迭代和交付效率。

服务交付之后需要部署运行，对微服务来说，它们运行期间也是各自独立的。

微服务独立运行可以带来两个比较明显的好处，第一个就是可扩展性。我们可以快速地添加服务集群的实例，提升整个微服务集群的服务能力，而在传统 Monolith 模式下，为了能够提升服务能力，很多时候必须强化和扩展单一结点的服务能力来达成。如果单结点服务能力已经扩展到了极限，再寻求扩展的

话，就得从软件到硬件整体进行重构。

软件行业有句话："Threads don't scale, Processes do!"，很明确地道出了原来 Monolith 服务与微服务在扩展（Scale）层面的差异。

对于 Java 开发者来说，早些年（当然现在也依然存在），我们遵循 Java EE 规范开发的 Web 应用，都需要以 WAR 包的形式部署到 TOMCAT、Jetty、RESIN 等 Web 容器中运行，即使每个 WAR 包提供的都是独立的微服务，但因为它们都是统一部署运行在一个 Web 容器中，所以扩展能力受限于 Web 容器作为一个进程（process）的现状。无论如何调整 Web 容器内部实现的线程（thread）设置，还是会受限于 Web 容器整体的扩展能力。所以，现在很多情况下，大家都是一个 TOMCAT 只部署一个 WAR，然后通过复制和扩展多个 TOMCAT 实例来扩展整个应用服务集群。

当然，说到在 TOMCAT 实例中只部署一个 WAR 包这样的做法，实际上不单单只是因为扩展的因素，还涉及微服务运行期间给我们带来的第二个好处，即隔离性。

隔离性实际上是可扩展性的基础，当我们将每个微服务都隔离为独立的运行单元之后，任何一个或者多个微服务的失败都将只影响自己或者少量其他微服务，而不会大面积地波及整个服务运行体系。在架构设计上有一种实践模式，即隔板模式（Bulkhead Pattern），这种架构设计模式的首要目的就是为了隔离系统中的各个功能单元和实体，使得系统不会因为一个单元或者服务的失败而导致整体失败。这种思路在造船行业、兵工行业都有类似的应用场景。现在任何大型船舶在设计上都会有隔舱，目的就是即使有少量进水，也可以只将进水部位隔离在小范围，不会扩散而导致船舶大面积进水，从而沉没。当年泰坦尼克号虽然沉了，但不意味着他们没有做隔舱设计，只能说，伤害度已经远远超出隔舱可以提供的基础保障范围。在坦克的设计上，现在一般也会将弹药舱和乘员舱隔离，从而可以保障当坦克受创之后，将伤害尽量限定在指定区域，尽量减少对车乘成员的伤害。

前面我们提到，现在大家基本上弱化了 Java EE 的 Web 容器早期采用的"一个 Web 容器部署多个 WAR 包"的做法，转而使用"一个 Web 容器只部署一个 WAR 包"的做法，这实际上正是综合考虑了 Web 容器的设计和实现现状与真实需求之后做出的合理实践选择。这些 Web 容器内部大多通过类加载器

（Classloader）以及线程来实现一定程度上的依赖和功能隔离，但这些机制从基因上决定了这些做法不是最好的隔离手段。而进程（Process）拥有天然的隔离特性，所以，一个 WAR 包只部署运行在一个 Web 容器进程中才是最好的隔离方式。

现在回想一下，好像自从各个微服务打响独立战争并且独立之后，无论从哪个层面来看，各自“活”得都挺好。

1.3.2 多语言生态

微服务独立之后，给了对应的团队和组织快速迭代和交付的能力，同时，也给团队和组织带来了更多的灵活性，实际上，对应交付不同微服务的团队或者组织来说，现在可以基于不同的计算机语言生态构建这些微服务，如图 1-2 所示。

微服务的提供者既可以使用 Java 或者 Go 等静态语言完成微服务的开发和交付，也可以使用 Python 或者 Ruby 等动态语言完成微服务的开发和交付，对于团队内部拥有繁荣且有差异的语言文化来说，多语言生态下的微服务开发和交付将可以最大化的发挥团队和组织内部各成员的优势。当然，对于多语言生态下的微服务研发来说，有一点需要注意：为了让服务的访问者可以用统一的接口访问所有这些用不同语言开发和交互的微服务，应该尽量统一微服务的服务接口和协议。

在微服务的生态下，互通性应该是需要重点关注的因素，没有互通，不但服务的访问者和用户无法很好地使用这些微服务，微服务和微服务之间也无法相互信赖和互助，这将大大损耗微服务研发体系带来的诸多好处，而多语言生态也会变成一种障碍和负累，而不是益处。

记得时任黑猫宅急便社长的小仓昌男在其所著的《黑猫宅急便的经营学》中提到一个故事，日本国铁曾经采用不同于国际标准的集装箱和铁路规格，然后发现货物的运输效率很低，经过考察发现，原来是货物从国际标准集装箱卸载之后，在通过日本国铁运输之前，需要先拆箱，重新装入日本国铁规格的集装箱，然后装载到日本国铁上进行运输。但是，如果日本国铁采用国际标准的集装箱规格，那么货物集装箱从远洋轮船上卸载之后就可以直接装上国铁，这将大大加快运输效率（日本，国铁改革后也证明确实如此）。日本国铁在前期采用私有方案时，只关注了自己的利益和效率，舍弃了互通，也带来了效率的低下。所以，在开发和交付微服务的时候，尤其是在多语言生态下开发和交付微

服务，我们从一开始就要将互通性作为首要考虑因素，从而不会因为执迷于某些服务或者系统的单点效率而失去了整个微服务体系的整体效率。

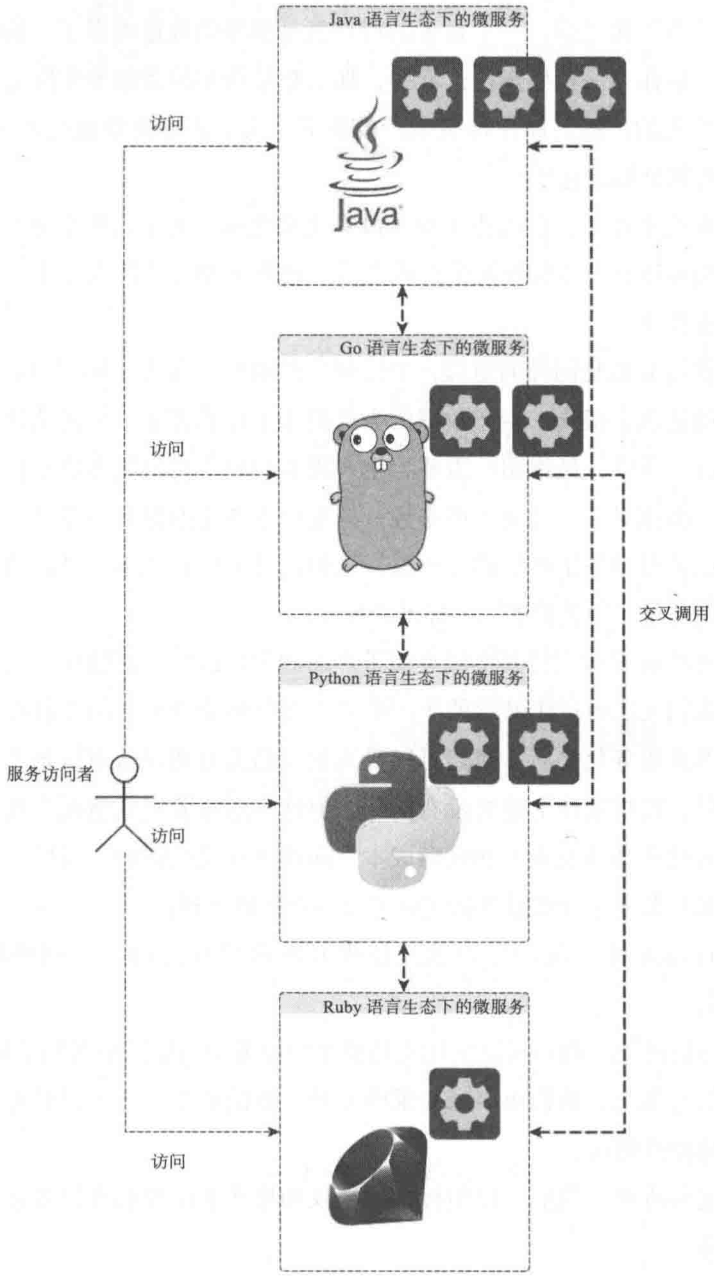


图 1-2　多语言的微服务生态

1.4 微服务会带来哪些挑战

微服务给我们带来的并非只有好处，还有相应的一些挑战。

服务“微”化之后，一个显著的特点就是服务的数量增多了。如果将软件开发和交付也作为一种生产模式看待，那么数量众多的微服务实际上就类似于传统生产线上的产品，而在传统生产模型下，为了能够高效地生产大量产品，通常采用的就是标准化生产。

比如在汽车产业，在福特 T 型车没有出来之前，大多汽车企业的生产效率都不高，而福特在引入标准化生产线之后，福特 T 型车得以大量生产并以低成本优势快速普及。

在其他行业也是同样的道理，个性化生产虽然会深得个别用户的喜欢，但生产成本通常也会很高，生产效率因为受限于个性化需求，也无法从“熟能生巧”中获益，所以，最终用户需要为生产成本和效率付出更多的溢价才能获得最终产品。而相对于个性化生产来说，标准化生产走的是另一条路，通过生产标准产品，使得整条生产链路可重复，从而提升了生产效率，可以为更广层面的用户提供大量“物美价廉”的标准产品。

微服务的研发和交付其实就类似于产品的生产链路，而数量大这一特点则决定了，我们无法通过个性化的生产模式来支撑整个微服务的交付链路和研发体系，虽然微服务化之后，我们可以投入相应的人力和团队对应各个微服务的开发和交付，可扩展性上绝对没有问题，但这不意味着现实情况下我们就能这样做，因为这些都涉及人力和资源成本，而这往往是受限的。所以，使用标准化的思路来开发和交付微服务就变成了自然而然的选择：

- 通过标准化，我们可以重复使用开发阶段打造的一系列环境和工具支持。
- 通过标准化，我们可以复用支持整个微服务交付链路的各项基础设施。
- 通过标准化，我们可以减少采购差异导致的成本上升，同时更加高效地利用硬件资源。
- 通过标准化，我们可以用标准的协议和格式来治理和维护数量庞大的微服务。

如果你还对使用标准化的思路来构建微服务体系存有疑惑，那么，不妨再

结合微服务的多语言生态特性思考一番：

- 增加一种语言生态用于微服务的开发和交付，我们是否要围绕着这种语言生态和微服务的需求重新搭建一套研发 / 测试环境？
- 我们是否还要围绕着这种语言生态打造一系列的工具来提升日常开发的效率？
- 增加一种语言生态，我们是不是还要围绕这种语言生态搭建一套针对微服务的交付链路基础设施？
- 增加一种语言生态，我们是否还要围绕它提供特定的硬件环境以及运维支撑工具和平台？

多语言生态虽然灵活度高了，不同语种和思路的团队成员也能够百花齐放了，但是不是也同样带来了以上一系列的成本？

所以，很多事情你能做，并不意味着你一定要做。适度的收缩语言生态的选择范围，并围绕主要的语言生态构建一套标准化的微服务交付体系，或许是更为合理的做法。

要实施高效可重复的标准化微服务生产，我们需要有类似传统行业生产线的基础设施。否则，高效可重复的开发和交付大量的微服务就无从谈起，所以，完备的微服务研发和交付体系基础设施建设就成为了实施微服务的终极挑战。一个公司或者组织要很好地或者说成熟地实施微服务化战略，为交付链路提供完备支撑的基础设施建设必不可少！

1.5　本章小结

在带领大家探索本书的主角 SpringBoot 微框架之前，本章首先为大家介绍了 SpringBoot 微框架服务的核心场景，即微服务。然后一起探索了微服务的概念以及由来，并探讨了微服务可以为我们带来哪些好处，以及同时又为我们带来哪些挑战。

总的来说，微服务化虽然是当下流行的趋势，但并非任何场景都合适，我们还是要审慎地在“大一统”（Monolith）服务架构和微服务架构之间做出选择，而一旦确定选择了微服务化之路，那么，就应该围绕团队和组织的主要语言生态以及微服务方向积极探索高效的微服务开发和交付模式。

SpringBoot 微框架实际上就是为 Java 语言生态而生的一种微服务最佳实践，在第 2 章中我们将从回顾 SpringBoot 的起源开始，逐步揭开 SpringBoot 微框架的神秘面纱。

第 2 章 Chapter 2

饮水思源：回顾与探索 Spring 框架的本质

SpringBoot 框架的命名关键在“ Boot”上，或许 Boot Spring 更能说明这个微框架设计的初衷，也就是快速启动一个 Spring 应用！

所以，自始至终，SpringBoot 框架都是为了能够帮助使用 Spring 框架的开发者快速高效地构建一个个基于 Spring 框架以及 Spring 生态体系的应用解决方案。要深刻理解 SpringBoot 框架，首先我们需要深刻理解 Spring 框架，所以让我们先来读读历史吧！

2.1 Spring 框架的起源

虽然笔者在自己的上一本著作《 Spring 揭秘》中对 Spring 框架进行了十分详尽的介绍和剖析，但这里还是要再啰嗦几句。

Spring 框架诞生于“黑暗”的 EJB 1 的时代（如果你没有听说过，恭喜你，说明你还年轻），那是一个 J2EE 规范统治的时代，基于各种容器和 J2EE 规范的软件解决方案是唯一的“正道”，沉重的研发模式和生态让那个时代的开发者痛苦不堪。随着经典巨著《 Expert One-on-One J2EE Design and Development 》的诞生，重规范时代终于迎来了一线曙光，该书的作者 Rod Johnson 在书中阐

述了轻量级框架的研发理念，对原有笨重的规范进行了抨击，并基于书中的理念推出了最初版的 Spring 框架，并延续至今已达 10 多年之久。

Spring 框架是构建高效 Java 研发体系的一种最佳实践，它通过一系列统一而简洁的设计，为广大 Java 开发者开拓了一条光明的 Java 应用最佳实践之路。

大家熟知的 Spring IoC 与 AOP 自不必说，Spring 更是对 Java 应用开发中常用的技术进行了合理的设计和封装，使得 Java 应用开发者可以避免昔日因 API 和系统设计不当而易犯的错误，又能够高效地完成相应问题领域的研发工作，真可说是 Java 开发必备良器！

当然，因为这不是一本专门介绍 Spring 框架的书，所以，这里不会详细展开对 Spring 框架的细节回顾。不过，一些核心的实践以及与 SpringBoot 相关的概念，还是有必要说在前的，比如 Spring IoC！

2.2 Spring IoC 其实很简单

有部分 Java 开发者对 IoC（Inversion Of Control）和 DI（Dependency Injection）的概念有些混淆，认为二者是对等的，实际上我在之前的著作中已经说过了，IoC 其实有两种方式，一种就是 DI，而另一种是 DL，即 Dependency Lookup（依赖查找），前者是当前软件实体被动接受其依赖的其他组件被 IoC 容器注入，而后者则是当前软件实体主动去某个服务注册地查找其依赖的那些服务，概念之间的关系如图 2-1 所示可能更贴切些。

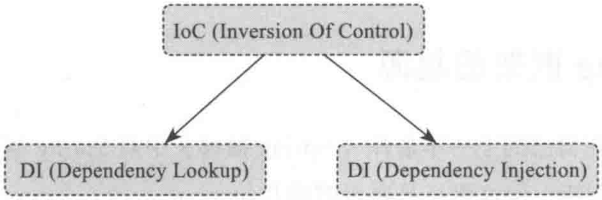


图 2-1 IoC 相关概念示意图

我们通常提到的 Spring IoC，实际上是指 Spring 框架提供的 IoC 容器实现（IoC Container），而使用 Spring IoC 容器的一个典型代码片段就是：

```
public class App {
    public static void main(String[] args) {
```

```
        ApplicationContext context = new FileSystemXmlApplication-
Context("...");
        // ...
        MockService service = context.getBean(MockService.class);
        service.doSomething();
    }
}
```

任何一个使用 Spring 框架构建的独立的 Java 应用（Standalone Java Application），通常都会存在一行类似于“context.getBean(..);”的代码，实际上，这行代码做的就是 DL 的工作，而构建的任何一种 IoC 容器背后（比如 BeanFactory 或者 ApplicationContext）发生的事情，则更多是 DI 的过程（也可能有部分 DL 的逻辑用于对接遗留系统）。

Spring 的 IoC 容器中发生的事情其实也很简单，总结下来即两个阶段：

（1）采摘和收集“咖啡豆”(bean)

（2）研磨和烹饪咖啡

哦，不对，这是一本技术书，差点儿写成咖啡文化杂志。

那我们还是回过头来继续说 Spring IoC 容器的依赖注入流程吧！Spring IoC 容器的依赖注入工作可以分为两个阶段：

阶段一：收集和注册

第一个阶段可以认为是构建和收集 bean 定义的阶段，在这个阶段中，我们可以通过 XML 或者 Java 代码的方式定义一些 bean，然后通过手动组装或者让容器基于某些机制自动扫描的形式，将这些 bean 定义收集到 IoC 容器中。

假设我们以 XML 配置的形式来收集并注册单一 bean，一般形式如下：

```
<bean id="mockService" class="..MockServiceImpl">
    ...
</bean>
```

如果嫌逐个收集 bean 定义麻烦，想批量地收集并注册到 IoC 容器中，我们也可以通过 XML Schema 形式的配置进行批量扫描并采集和注册：

```
<context:component-scan base-package="com.keevol">
```

注意　基于 JavaConfig 形式的收集和注册，不管是单一还是批量，后面我们都会单独提及。

阶段二：分析和组装

当第一阶段工作完成后，我们可以先暂且认为 IoC 容器中充斥着一个个独立的 bean，它们之间没有任何关系。但实际上，它们之间是有依赖关系的，所以，IoC 容器在第二阶段要干的事情就是分析这些已经在 IoC 容器之中的 bean，然后根据它们之间的依赖关系先后组装它们。如果 IoC 容器发现某个 bean 依赖另一个 bean，它就会将这另一个 bean 注入给依赖它的那个 bean，直到所有 bean 的依赖都注入完成，所有 bean 都“整装待发”，整个 IoC 容器的工作即算完成。

至于分析和组装的依据，Spring 框架最早是通过 XML 配置文件的形式来描述 bean 与 bean 之间的关系的，随着 Java 业界研发技术和理念的转变，基于 Java 代码和 Annotation 元信息的描述方式也日渐兴盛（比如 @Autowired 和 @Inject），但不管使用哪种方式，都只是为了简化绑定逻辑描述的各种“表象”，最终都是为本阶段的最终目的服务。

提示 很多 Java 开发者一定认为 spring 的 XML 配置文件是一种配置（Configuration），但本质上，这些配置文件更应该是一种代码形式，XML 在这里其实可以看作一种 DSL，它用来表述的是 bean 与 bean 之间的依赖绑定关系，诸君还记得没有 IoC 容器的年代要自己写代码新建（new）对象并配置（set）依赖的吧？

2.3 了解一点儿 JavaConfig

Java 5 的推出，加上当年基于纯 Java Annotation 的依赖注入框架 Guice 的出现，使得 Spring 框架及其社区也“顺应民意”，推出并持续完善了基于 Java 代码和 Annotation 元信息的依赖关系绑定描述方式，即 JavaConfig 项目。

基于 JavaConfig 方式的依赖关系绑定描述基本上映射了最早的基于 XML 的配置方式，比如：

（1）表达形式层面

基于 XML 的配置方式是这样的：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.
xsd http://www.springframework.org/schema/context http://www.
springframework.org/schema/context/spring-context.xsd">
    <!-- bean 定义 -->
</beans>
```

而基于 JavaConfig 的配置方式是这样的：

```
@Configuration
public class MockConfiguration{
    // bean 定义
}
```

任何一个标注了 @Configuration 的 Java 类定义都是一个 JavaConfig 配置类。

（2）注册 bean 定义层面

基于 XML 的配置形式是这样的：

```
<bean id="mockService" class="..MockServiceImpl">
    ...
</bean>
```

而基于 JavaConfig 的配置形式是这样的：

```
@Configuration
public class MockConfiguration {
    @Bean
    public MockService mockService() {
        return new MockServiceImpl();
    }
}
```

任何一个标注了 @Bean 的方法，其返回值将作为一个 bean 定义注册到 Spring 的 IoC 容器，方法名将默认成为该 bean 定义的 id。

（3）表达依赖注入关系层面

为了表达 bean 与 bean 之间的依赖关系，在 XML 形式中一般是这样的：

```
<bean id="mockService" class="..MockServiceImpl">
    <property name="dependencyService" ref="dependencyService"/>
```

```
</bean>

<bean id="dependencyService" class="DependencyServiceImpl"/>
```

而在 JavaConfig 中则是这样的：

```
@Configuration
public class MockConfiguration {

    @Bean
    public MockService mockService() {
        return new MockServiceImpl(dependencyService());
    }

    @Bean
    public DependencyService dependencyService() {
        return new DependencyServiceImpl();
    }
}
```

如果一个 bean 的定义依赖其他 bean，则直接调用对应 JavaConfig 类中依赖 bean 的创建方法就可以了。

注意 在 JavaConfig 形式的依赖注入过程中，我们使用方法调用的形式注入依赖，如果这个方法返回的对象实例只被一个 bean 依赖注入，那也还好，如果多于一个 bean 需要依赖这个方法调用返回的对象实例，那是不是意味着我们就会创建多个同一类型的对象实例？

从代码表述的逻辑来看，直觉上应该是会创建多个同一类型的对象实例，但实际上最终结果却不是这样，依赖注入的都是同一个 Singleton 的对象实例，那这是如何做到的？

笔者一开始以为 Spring 框架会通过解析 JavaConfig 的代码结构，然后通过解析器转换加上反射等方式完成这一目的，但实际上 Spring 框架的设计和实现者采用了另一种更通用的方式，这在 Spring 的参考文档中有说明，即通过拦截配置类的方法调用来避免多次初始化同一类型对象的问题，一旦拥有拦截逻辑的子类发现当前方法没有对应的类型实例时才会去请求父类的同一方法来初始化对象实例，否则直接返回之前的对象实例。

所以，原来 Spring IoC 容器中有的特性（features）在 JavaConfig 中都可以表述，只是换了一种形式而已，而且，通过声明相应的 Java Annotation 反而“内聚”一处，变得更加简洁明了了。

2.3.1 那些高曝光率的 Annotation

至于 @Configuration，我想前面已经提及过了，这里不再赘述，下面我们看几个其他比较常见的 Annotation，便于为后面更好地理解 SpringBoot 框架的奥秘做准备。

1. @ComponentScan

@ComponentScan 对应 XML 配置形式中的 <context:component-scan> 元素，用于配合一些元信息 Java Annotation，比如 @Component 和 @Repository 等，将标注了这些元信息 Annotation 的 bean 定义类批量采集到 Spring 的 IoC 容器中。

我们可以通过 basePackages 等属性来细粒度地定制 @ComponentScan 自动扫描的范围，如果不指定，则默认 Spring 框架实现会从声明 @ComponentScan 所在类的 package 进行扫描。

@ComponentScan 是 SpringBoot 框架魔法得以实现的一个关键组件，大家可以重点关注，我们后面还会遇到它。

2. @PropertySource 与 @PropertySources

@PropertySource 用于从某些地方加载 *.properties 文件内容，并将其中的属性加载到 IoC 容器中，便于填充一些 bean 定义属性的占位符（placeholder），当然，这需要 PropertySourcesPlaceholderConfigurer 的配合。

如果我们使用 Java 8 或者更高版本开发（本书写作期间 Java 9 还没发布），那么，我们可以并行声明多个 @PropertySource：

```
@Configuration
@PropertySource("classpath:1.properties")
@PropertySource("classpath:2.properties")
@PropertySource("...")
public class XConfiguration{
    ...
}
```

如果我们使用低于 Java 8 版本的 Java 开发 Spring 应用，又想声明多个 @PropertySource，则需要借助 @PropertySources 的帮助了：

```
@PropertySources({
        @PropertySource("classpath:1.properties"),
        @PropertySource("classpath:2.properties"),
        ...
})
public class XConfiguration{
    ...
}
```

3. @Import 与 @ImportResource

在 XML 形式的配置中，我们通过 <import resource="XXX.xml"/> 的形式将多个分开的容器配置合到一个配置中，在 JavaConfig 形式的配置中，我们则使用 @Import 这个 Annotation 完成同样目的：

```
@Configuration
@Import(MockConfiguration.class)
public class XConfiguration {
    ...
}
```

@Import 只负责引入 JavaConfig 形式定义的 IoC 容器配置，如果有一些遗留的配置或者遗留系统需要以 XML 形式来配置（比如 dubbo 框架），我们依然可以通过 @ImportResource 将它们一起合并到当前 JavaConfig 配置的容器中：

```
@Configuration
@Import(MockConfiguration.class)
@ImportResource("...")
public class XConfiguration {
    ...
}
```

2.4 本章小结

“磨刀不误砍柴工”，本章我们主要回顾了一下 Spring 框架的历史，并对 Spring 框架的一些核心功能和特性进行了精炼的剖析，在把我们的思维之刀磨砺快了之后，让我们开始解一下 SpringBoot 这头小牛儿吧！

第 3 章 Chapter 3

SpringBoot 的工作机制

我们说 SpringBoot 是 Spring 框架对"约定优先于配置（Convention Over Configuration)"理念的最佳实践的产物，一个典型的 SpringBoot 应用本质上其实就是一个基于 Spring 框架的应用，而如果大家对 Spring 框架已经了如指掌，那么，在我们一步步揭开 SpringBoot 微框架的面纱之后，大家就会发现"阳光之下，并无新事"。不信？那我们一起走着瞧呗！

3.1 SpringBoot 初体验

一个典型的 SpringBoot 应用长什么样子呢？如果我们使用 http://start.spring.io/ 创建一个最简单的依赖 Web 模块的 SpringBoot 应用，一般情况下，我们会得到一个 SpringBoot 应用的启动类，如下面代码所示：

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
```

```
    }
}
```

所有的 SpringBoot 无论怎么定制，本质上与上面的启动类代码是一样的，而以上代码示例中，Annotation 定义（@SpringBootApplication）和类定义（SpringApplication.run）最为耀眼，那么，要揭开 SpringBoot 应用的奥秘，很明显的，我们只要先从这两位开始就可以了。

3.2 @SpringBootApplication 背后的秘密

@SpringBootApplication 是一个“三体”结构，实际上它是一个复合 Annotation：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication{
    ...
}
```

虽然它的定义使用了多个 Annotation 进行元信息标注，但实际上对于 SpringBoot 应用来说，重要的只有三个 Annotation，而“三体”结构实际上指的就是这三个 Annotation：

- @Configuration
- @EnableAutoConfiguration
- @ComponentScan

所以，如果我们使用如下的 SpringBoot 启动类，整个 SpringBoot 应用依然可以与之前的启动类功能对等：

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class DemoApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

但每次都写三个 Annotation 显然过于繁琐，所以写一个 @SpringBoot-Application 这样的一站式复合 Annotation 显然更方便些。

3.2.1 @Configuration 创世纪

这里的 @Configuration 对我们来说并不陌生，它就是 JavaConfig 形式的 Spring IoC 容器的配置类使用的那个 @Configuration，既然 SpringBoot 应用骨子里就是一个 Spring 应用，那么，自然也需要加载某个 IoC 容器的配置，而 SpringBoot 社区推荐使用基于 JavaConfig 的配置形式，所以，很明显，这里的启动类标注了 @Configuration 之后，本身其实也是一个 IoC 容器的配置类！

很多 SpringBoot 的代码示例都喜欢在启动类上直接标注 @Configuration 或者 @SpringBootApplication，对于初接触 SpringBoot 的开发者来说，其实这种做法不便于理解，如果我们将上面的 SpringBoot 启动类拆分为两个独立的 Java 类，整个形势就明朗了：

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class DemoConfiguration {
    @Bean
    public Controller controller() {
        return new Controller();
    }
}


public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoConfiguration.class, args);
    }
}
```

所以，启动类 DemoApplication 其实就是一个标准的 Standalone 类型 Java 程序的 main 函数启动类，没有什么特殊的。

而 @Configuration 标注的 DemoConfiguration 定义其实也是一个普通的 JavaConfig 形式的 IoC 容器配置类，没啥新东西，全是 Spring 框架里的概念！

3.2.2 @EnableAutoConfiguration 的功效

@EnableAutoConfiguration 其实也没啥“创意”，各位是否还记得 Spring 框架提供的各种名字为 @Enable 开头的 Annotation 定义？比如 @EnableScheduling、@EnableCaching、@EnableMBeanExport 等，@EnableAutoConfiguration 的理念和“做事方式”其实一脉相承，简单概括一下就是，借助 @Import 的支持，收集和注册特定场景相关的 bean 定义：

- @EnableScheduling 是通过 @Import 将 Spring 调度框架相关的 bean 定义都加载到 IoC 容器。
- @EnableMBeanExport 是通过 @Import 将 JMX 相关的 bean 定义加载到 IoC 容器。

而 @EnableAutoConfiguration 也是借助 @Import 的帮助，将所有符合自动配置条件的 bean 定义加载到 IoC 容器，仅此而已！

@EnableAutoConfiguration 作为一个复合 Annotation，其自身定义关键信息如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    ...
}
```

其中，最关键的要属 @Import(EnableAutoConfigurationImportSelector.class)，借助 EnableAutoConfigurationImportSelector，@EnableAutoConfiguration 可以帮助 SpringBoot 应用将所有符合条件的 @Configuration 配置都加载到当前 SpringBoot 创建并使用的 IoC 容器，就跟一只“八爪鱼”一样（如图 3-1 所示）。

借助于 Spring 框架原有的一个工具类：SpringFactoriesLoader 的支持，@EnableAutoConfiguration 可以“智能”地自动配置功效才得以大功告成！

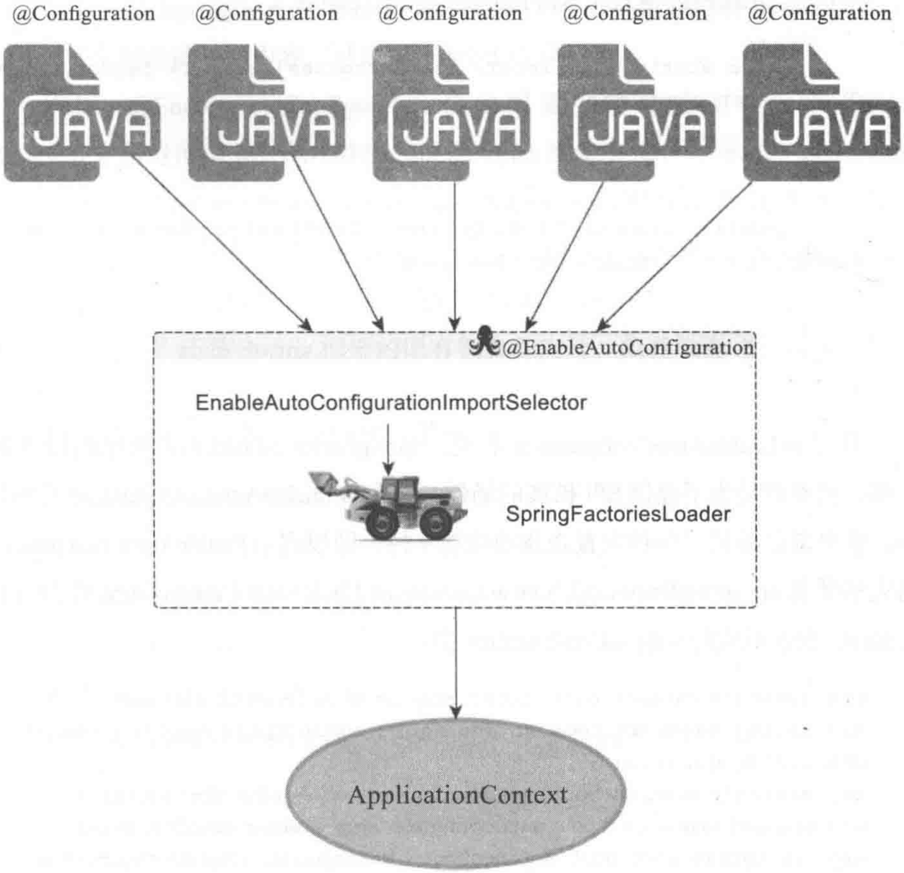


图 3-1 EnableAutoConfiguration 得以生效的关键组件关系图

自动配置的幕后英雄：SpringFactoriesLoader 详解

SpringFactoriesLoader 属于 Spring 框架私有的一种扩展方案（类似于 Java 的 SPI 方案 java.util.ServiceLoader），其主要功能就是从指定的配置文件 META-INF/spring.factories 加载配置，spring.factories 是一个典型的 java properties 文件，配置的格式为 Key = Value 形式，只不过 Key 和 Value 都是 Java 类型的完整类名（Fully qualified name），比如[⊖]：

```
example.MyService=example.MyServiceImpl1,example.MyServiceImpl2
```

然后框架就可以根据某个类型作为 Key 来查找对应的类型名称列表了：

⊖ 摘自 SpringFactoriesLoader 的 Javadoc。

```
public abstract class SpringFactoriesLoader {
    // ...
    public static <T> List<T> loadFactories(Class<T> factoryClass,
ClassLoader classLoader) {
        ...
    }

    public static List<String> loadFactoryNames(Class<?>
factoryClass, ClassLoader classLoader) {
        ...
    }
    // ...
}
```

对于 @EnableAutoConfiguration 来说，SpringFactoriesLoader 的用途稍微不同一些，其本意是为了提供 SPI 扩展的场景，而在 @EnableAutoConfiguration 的场景中，它更多是提供了一种配置查找的功能支持，即根据 @EnableAutoConfiguration 的完整类名 org.springframework.boot.autoconfigure.EnableAutoConfiguration 作为查找的 Key，获取对应的一组 @Configuration 类：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdmin-
JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.PropertyPlaceholderAuto-
Configuration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAuto-
Configuration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationProperties-
AutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceException-
TranslationAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.Cassandra-
DataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.Cassandra-
RepositoriesAutoConfiguration,\
...
```

以上是从 SpringBoot 的 autoconfigure 依赖包中的 META-INF/spring.factories 配置文件中摘录的一段内容，可以很好地说明问题。

所以，@EnableAutoConfiguration 自动配置的魔法其实就变成了：从 classpath 中搜寻所有 META-INF/spring.factories 配置文件，并将其中 org.springframework.boot.autoconfigure.EnableAutoConfiguration 对应的配置项通过反射（Java Reflection）实例化为对应的标注了 @Configuration 的 JavaConfig 形式的 IoC 容器配置类，然后汇总为一个并加载到 IoC 容器。

目前为止，还是 Spring 框架的原有概念和支持，依然没有“新鲜事”！

3.2.3 可有可无的 @ComponentScan

为啥说 @ComponentScan 是可有可无的？因为原则上来说，作为 Spring 框架里的“老一辈革命家”，@ComponentScan 的功能其实就是自动扫描并加载符合条件的组件或 bean 定义，最终将这些 bean 定义加载到容器中。加载 bean 定义到 Spring 的 IoC 容器，我们可以手工单个注册，不一定非要通过批量的自动扫描完成，所以说 @ComponentScan 是可有可无的。

对于 SpringBoot 应用来说，同样如此，比如我们本章的启动类：

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

如果我们当前应用没有任何 bean 定义需要通过 @ComponentScan 加载到当前 SpringBoot 应用对应使用的 IoC 容器，那么，除去 @ComponentScan 的声明，当前 SpringBoot 应用依然可以照常运行，功能对等！

看，还是没有啥新东西！

3.3 SpringApplication：SpringBoot 程序启动的一站式解决方案

如果非说 SpringBoot 微框架提供了点儿自己特有的东西，在核心类层面（各种场景下的自动配置一站式插拔模块，我们下一章再重点介绍），也就是 SpringApplication 了。

SpringApplication 将一个典型的 Spring 应用启动的流程“模板化”（这里是动词），在没有特殊需求的情况下，默认模板化后的执行流程就可以满足需求了；但有特殊需求也没关系，SpringApplication 在合适的流程结点开放了一系列不同类型的扩展点，我们可以通过这些扩展点对 SpringBoot 程序的启动和关闭过程进行扩展。

最“肤浅”的扩展或者配置是 SpringApplication 通过一系列设置方法（setters）开放的定制方式，比如，我们之前的启动类的 main 方法中只有一句：

```
SpringApplication.run(DemoApplication.class, args);
```

但如果我们想通过 SpringApplication 的一系列设置方法来扩展启动行为，则可以用如下方式进行：

```
public class DemoApplication {

    public static void main(String[] args) {
        // SpringApplication.run(DemoConfiguration.class, args);

         SpringApplication bootstrap = new SpringApplication(Demo-
Configuration.class);
        bootstrap.setBanner(new Banner() {
            @Override
            public void printBanner(Environment environment, Class<?>
aClass, PrintStream printStream) {
                // 比如打印一个我们喜欢的 ASCII Arts 字符画
            }
        });
        bootstrap.setBannerMode(Banner.Mode.CONSOLE);
        // 其他定制设置 ...
        bootstrap.run(args);
    }
}
```

> **提示** 设置自定义 banner 最简单的方式其实是把 ASCII Art 字符画放到一个资源文件，然后通过 ResourceBanner 来加载：bootstrap.setBanner(new ResourceBanner(new ClassPathResource("banner.txt")));

大部分情况下，SpringApplication 已经提供了很好的默认设置，所以，我们不再对这些表层进行探究了，因为对表层之下的东西进行探究才是我们的最终目的。

3.3.1 深入探索 SpringApplication 执行流程

SpringApplication 的 run 方法的实现是我们本次旅程的主要线路，该方法的主要流程大体可以归纳如下[⊖]：

1）如果我们使用的是 SpringApplication 的静态 run 方法，那么，这个方法里面首先需要创建一个 SpringApplication 对象实例，然后调用这个创建好的 SpringApplication 的实例 run 方法。在 SpringApplication 实例初始化的时候，它会提前做几件事情：

- 根据 classpath 里面是否存在某个特征类（org.springframework.web.context.ConfigurableWebApplicationContext）来决定是否应该创建一个为 Web 应用使用的 ApplicationContext 类型，还是应该创建一个标准 Standalone 应用使用的 ApplicationContext 类型。
- 使用 SpringFactoriesLoader 在应用的 classpath 中查找并加载所有可用的 ApplicationContextInitializer。
- 使用 SpringFactoriesLoader 在应用的 classpath 中查找并加载所有可用的 ApplicationListener。
- 推断并设置 main 方法的定义类。

2）SpringApplication 实例初始化完成并且完成设置后，就开始执行 run 方法的逻辑了，方法执行伊始，首先遍历执行所有通过 SpringFactoriesLoader 可以查找到并加载的 SpringApplicationRunListener，调用它们的 started() 方法，告诉这些 SpringApplicationRunListener，“嘿，SpringBoot 应用要开始执行咯！”。

3）创建并配置当前 SpringBoot 应用将要使用的 Environment（包括配置要

⊖ 本流程说明参考的是 SpringBoot 1.2.6 版本代码的实现。

使用的 PropertySource 以及 Profile）。

4）遍历调用所有 SpringApplicationRunListener 的 environmentPrepared() 的方法，告诉它们："当前 SpringBoot 应用使用的 Environment 准备好咯！"。

5）如果 SpringApplication 的 showBanner 属性被设置为 true，则打印 banner（SpringBoot 1.3.x 版本，这里应该是基于 Banner.Mode 决定 banner 的打印行为）。这一步的逻辑其实可以不关心，我认为唯一的用途就是"好玩"（Just For Fun）。

6）根据用户是否明确设置了 applicationContextClass 类型以及初始化阶段的推断结果，决定该为当前 SpringBoot 应用创建什么类型的 ApplicationContext 并创建完成，然后根据条件决定是否添加 ShutdownHook，决定是否使用自定义的 BeanNameGenerator，决定是否使用自定义的 ResourceLoader，当然，最重要的，将之前准备好的 Environment 设置给创建好的 ApplicationContext 使用。

7）ApplicationContext 创建好之后，SpringApplication 会再次借助 Spring-FactoriesLoader，查找并加载 classpath 中所有可用的 ApplicationContext-Initializer，然后遍历调用这些 ApplicationContextInitializer 的 initialize (applicationContext) 方法来对已经创建好的 ApplicationContext 进行进一步的处理。

8）遍历调用所有 SpringApplicationRunListener 的 contextPrepared() 方法，通知它们："SpringBoot 应用使用的 ApplicationContext 准备好啦！"

9）最核心的一步，将之前通过 @EnableAutoConfiguration 获取的所有配置以及其他形式的 IoC 容器配置加载到已经准备完毕的 ApplicationContext。

10）遍历调用所有 SpringApplicationRunListener 的 contextLoaded() 方法，告知所有 SpringApplicationRunListener，ApplicationContext" 装填完毕 "!

11）调用 ApplicationContext 的 refresh() 方法，完成 IoC 容器可用的最后一道工序。

12）查找当前 ApplicationContext 中是否注册有 CommandLineRunner，如果有，则遍历执行它们。

13）正常情况下，遍历执行 SpringApplicationRunListener 的 finished() 方法，告知它们："搞定！"。（如果整个过程出现异常，则依然调用所有 SpringApplicationRunListener 的 finished() 方法，只不过这种情况下会将异常信

息一并传入处理)。

至此，一个完整的 SpringBoot 应用启动完毕!

整个过程看起来冗长无比，但其实很多都是一些事件通知的扩展点，如果我们将这些逻辑暂时忽略，那么，其实整个 SpringBoot 应用启动的逻辑就可以压缩到极其精简的几步，如图 3-2 所示。

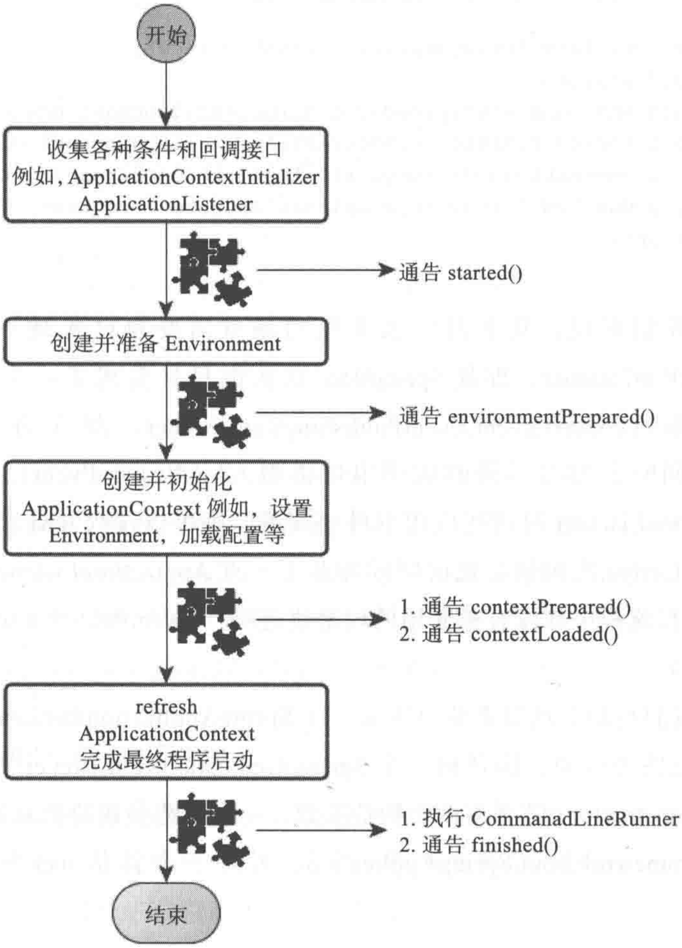


图 3-2 SpringBoot 应用启动步骤简要示意图

前后对比我们就可以发现，其实 SpringApplication 提供的这些各类扩展点近乎“喧宾夺主”，占据了一个 Spring 应用启动逻辑的大部分“江山”，除了初始化并准备好 ApplicationContext，剩下的大部分工作都是通过这些扩展点完成

的，所以，我们有必要对各类扩展点进行逐一剖析，以便在需要的时候可以信手拈来，为我所用。

3.3.2 SpringApplicationRunListener

SpringApplicationRunListener 是一个只有 SpringBoot 应用的 main 方法执行过程中接收不同执行时点事件通知的监听者：

```
public interface SpringApplicationRunListener {
    void started();
    void environmentPrepared(ConfigurableEnvironment environment);
    void contextPrepared(ConfigurableApplicationContext context);
    void contextLoaded(ConfigurableApplicationContext context);
    void finished(ConfigurableApplicationContext context, Throwable
exception);
}
```

对于我们来说，基本没什么常见的场景需要自己实现一个 SpringApplicationRunListener，即使 SpringBoot 默认也只是实现了一个 org.springframework.boot.context.event.EventPublishingRunListener，用于在 SpringBoot 启动的不同时点发布不同的应用事件类型（ApplicationEvent），如果有哪些 ApplicationListener 对这些应用事件感兴趣，则可以接收并处理。（还记得 SpringApplication 实例初始化的时候加载了一批 ApplicationListener，但是在 run 方法执行流程中却没有被使用的丝毫痕迹吗？EventPublishingRunListener 就是答案！）

假设我们真的有场景需要自定义一个 SpringApplicationRunListener 实现，那么有一点需要注意，即任何一个 SpringApplicationRunListener 实现类的构造方法（Constructor）需要有两个构造参数，一个构造参数的类型就是我们的 org.springframework.boot.SpringApplication，另外一个就是 args 参数列表的 String[]：

```
public class DemoSpringApplicationRunListener implements
SpringApplicationRunListener {
    @Override
    public void started() {
        // do whatever you want to do
    }
```

```
    @Override
    public void environmentPrepared(ConfigurableEnvironment environment) {
        // do whatever you want to do
    }

    @Override
    public void contextPrepared(ConfigurableApplicationContext context) {
        // do whatever you want to do
    }

    @Override
    public void contextLoaded(ConfigurableApplicationContext context) {
        // do whatever you want to do
    }

    @Override
    public void finished(ConfigurableApplicationContext context,
Throwable exception) {
        // do whatever you want to do
    }
}
```

之后，我们可以通过 SpringFactoriesLoader 立下的规矩，在当前 SpringBoot 应用的 classpath 下的 META-INF/spring.factories 文件中进行类似如下的配置：

```
org.springframework.boot.SpringApplicationRunListener=\
com.keevol.springboot.demo.DemoSpringApplicationRunListener
```

然后 SpringApplication 就会在运行的时候调用它啦！

3.3.3 ApplicationListener

ApplicationListener 其实是老面孔，属于 Spring 框架对 Java 中实现的监听者模式的一种框架实现，这里唯一值得着重强调的是，对于初次接触 SpringBoot，但对 Spring 框架本身又没有过多接触的开发者来说，可能会将这个名字与 SpringApplicationRunListener 混淆。

关于 ApplicationListener 我们就不做过多介绍了，如果感兴趣，请参考 Spring 框架相关的资料和书籍。

如果我们要为 SpringBoot 应用添加自定义的 ApplicationListener，有两种方式：

1）通过 SpringApplication.addListeners(..) 或者 SpringApplication.setListeners(..) 方法添加一个或者多个自定义的 ApplicationListener；

2）借助 SpringFactoriesLoader 机制，在 META-INF/spring.factories 文件中添加配置（以下代码是为 SpringBoot 默认注册的 ApplicationListener 配置）：

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.builder.ParentContextCloserApplicationListener,\
org.springframework.boot.cloudfoundry.VcapApplicationListener,\
org.springframework.boot.context.FileEncodingApplicationListener,\
org.springframework.boot.context.config.AnsiOutputApplicationListener,\
org.springframework.boot.context.config.ConfigFileApplicationListener,\
org.springframework.boot.context.config.DelegatingApplicationListener,\
org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicat-
ionListener,\
org.springframework.boot.logging.ClasspathLoggingApplicationListener,\
org.springframework.boot.logging.LoggingApplicationListener
```

关于 ApplicationListener，我们就说这些。

3.3.4 ApplicationContextInitializer

ApplicationContextInitializer 也是 Spring 框架原有的概念，这个类的主要目的就是在 ConfigurableApplicationContext 类型（或者子类型）的 ApplicationContext 做 refresh 之前，允许我们对 ConfigurableApplicationContext 的实例做进一步的设置或者处理。

实现一个 ApplicationContextInitializer 很简单，因为它只有一个方法需要实现：

```
public class DemoApplicationContextInitializer implements
ApplicationContextInitializer {
    @Override
    public void initialize(ConfigurableApplicationContext applicationContext)
{
        // do whatever you want with applicationContext,
        // e.g. applicationContext.registerShutdownHook();
    }
}
```

不过，一般情况下我们基本不会需要自定义一个 ApplicationContext-Initializer，即使 SpringBoot 框架默认也只是注册了三个实现：

```
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.context.ConfigurationWarningsApplication-
ContextInitializer,\
org.springframework.boot.context.ContextIdApplicationContextInitia-
lizer,\
org.springframework.boot.context.config.DelegatingApplicationContex-
tInitializer
```

如果我们真的需要自定义一个 ApplicationContextInitializer，那么只要像上面这样，通过 SpringFactoriesLoader 机制进行配置，或者通过 SpringApplication.addInitializers(..) 设置即可。

3.3.5 CommandLineRunner

CommandLineRunner 不是 Spring 框架原有的"宝贝"，它属于 SpringBoot 应用特定的回调扩展接口：

```
public interface CommandLineRunner {
    void run(String... args) throws Exception;
}
```

CommandLineRunner 需要大家关注的其实就两点：

1）所有 CommandLineRunner 的执行时点在 SpringBoot 应用的 ApplicationContext 完全初始化开始工作之后（可以认为是 main 方法执行完成之前最后一步）。

2）只要存在于当前 SpringBoot 应用的 ApplicationContext 中的任何 CommandLineRunner，都会被加载执行（不管你是手动注册这个 CommandLineRunner 到 IoC 容器，还是自动扫描进去的）。

与其他几个扩展点接口类型相似，建议 CommandLineRunner 的实现类使用 @org.springframework.core.annotation.Order 进行标注或者实现 org.springframework.core.Ordered 接口，便于对它们的执行顺序进行调整，这其实十分重要，我们不希望顺序不当的 CommandLineRunner 实现类阻塞了后面其他 CommandLineRunner 的执行。

CommandLineRunner 是很好的扩展接口，大家可以重点关注，我们在后面的扩展和微服务实践章节会再次遇到它。

3.4 再谈自动配置

此前我们讲到，@EnableAutoConfiguration 可以借助 SpringFactoriesLoader 这个特性将标注了 @Configuration 的 JavaConfig 类“一股脑儿”的汇总并加载到最终的 ApplicationContext，不过，这其实只是“简化版”的说明，实际上，基于 @EnableAutoConfiguration 的自动配置功能拥有更加强大的调控能力，通过配合比如基于条件的配置能力或者调整加载顺序，我们可以对自动配置进行更加细粒度的调整和控制。

3.4.1 基于条件的自动配置

基于条件的自动配置来源于 Spring 框架中“基于条件的配置”这一特性。在 Spring 框架中，我们可以使用 @Conditional 这个 Annotation 配合 @Configuration 或者 @Bean 等 Annotation 来干预一个配置或者 bean 定义是否能够生效，其最终实现的效果或者语义类似于如下伪代码：

```
if(符合@Conditional规定的条件){
    加载当前配置(enable current Configuration)或者注册当前bean定义;
}
```

要实现基于条件的配置，我们只要通过 @Conditional 指定自己的 Condition 实现类就可以了（可以应用于类型 Type 的标注或者方法 Method 的标注）：

```
@Conditional({MyCondition1.class, MyCondition2.class, ...})
```

最主要的是，@Conditional 可以作为一个 Meta Annotation 用来标注其他 Annotation 实现类，从而构建各色的复合 Annotation，比如 SpringBoot 的 autoconfigure 模块就基于这一优良的革命传统，实现了一批这样的 Annotation（位于 org.springframework.boot.autoconfigure.condition 包下）：

- @ConditionalOnClass
- @ConditionalOnBean
- @ConditionalOnMissingClass
- @ConditionalOnMissingBean
- @ConditionalOnProperty
- ……

有了这些复合Annotation的配合，我们就可以结合@EnableAuto-Configurationn实现基于条件的自动配置了。

SpringBoot能够风靡，很大一部分功劳需要归功于它预先提供的一系列自动配置的依赖模块，而这些依赖模块都是基于以上@Conditional复合Annotation实现的，这也意味着所有的这些依赖模块都是按需加载的，只有符合某些特定条件，这些依赖模块才会生效，这也就是我们所谓的“智能”自动配置。

3.4.2 调整自动配置的顺序

在实现自动配置的过程中，除了可以提供基于条件的配置，我们还可以对当前要提供的配置或者组件的加载顺序进行相应调整，从而让这些配置或者组件之间的依赖分析和组装可以顺利完成。

我们可以使用@org.springframework.boot.autoconfigure.AutoConfigureBefore或者@org.springframework.boot.autoconfigure.AutoConfigureAfter让当前配置或者组件在某个其他组件之前或者之后进行，比如，假设我们希望某些JMX操作相关的bean定义在MBeanServer配置完成之后进行，那么我们就可以提供一个类似如下的配置：

```
@Configuration
@AutoConfigureAfter(JmxAutoConfiguration.class)
public class AfterMBeanServerReadyConfiguration {
    @Autowired
    MBeanServer mBeanServer;

    // 通过@Bean添加必要的bean定义
}
```

3.5 本章小结

至此，我们对SpringBoot的核心组件完成了基本的剖析，综合来看，大部分的东西都是Spring框架背后原有的一些概念和实践方式，SpringBoot只是在这些概念和实践方式上对特定的场景实现进行了固化和升华，而也恰恰是这些固化让我们开发基于Spring框架的应用更加方便高效。

如果 SpringBoot 真有什么秘密可言的话，那也是 Spring 框架和 Spring 生态圈的秘密，如果大家对 Spring 框架和其生态圈已经了然于心，通过本章的讲解，相信 SpringBoot 对大家已经无甚神秘可言了吧！

通过适当的固化 Spring 应用的研发实践，SpringBoot 让 Spring 应用的研发更加高效，这并没有耗费 SpringBoot 设计者太多的智力和精力，背后也无太多高深的奥秘，但 SpringBoot 得以盛行并非靠拼“技术含量”，它默认提供的那些开箱即用的自动配置依赖模块才是让 Spring 开发者最受益的，下一章我们将就这些依赖模块进行相应的介绍，以帮助大家更好地了解并使用它们。

第 4 章 了解纷杂的 spring-boot-starter

Chapter 4

我认为，SpringBoot 微框架从两个主要层面影响 Spring 社区的开发者们：

1）基于 Spring 框架的“约定优先于配置（COC）”理念以及最佳实践之路。

2）提供了针对日常企业应用研发各种场景的 spring-boot-starter 自动配置依赖模块，如此多“开箱即用”的依赖模块，使得开发各种场景的 Spring 应用更加快速和高效。

SpringBoot 提供的这些“开箱即用”的依赖模块都约定以 spring-boot-starter- 作为命名的前缀，并且皆位于 org.springframework.boot 包或者命名空间下（虽然 SpringBoot 的官方参考文档中提到不建议大家使用 spring-boot-starter- 来命名自己写的类似的自动配置依赖模块，但实际上，配合不同的 groupId，这不应该是什么问题）。

如果我们访问 http://start.spring.io，并单击图 4-1 中的“ Switch to the full version”链接，就会发现 SpringBoot1.3.1 默认支持和提供了大约 80 多个自动配置依赖模块。

鉴于数量如此之多，并且也不是所有人都会在任何一个应用中用到所有，我只会就几个常见的通用 spring-boot-starter 模块进行讲解，希望大家可以举一反三，灵活应用所有日后工作过程中将会用到的那些 spring-boot-starter 模块。

所有的 spring-boot-starter 都有约定俗成的默认配置，但允许我们调整

这些配置以改变默认的配置行为，即“约定优先于配置”。在介绍相应的 spring-boot-starter 的默认配置（约定）以及可调整配置之前，我们有必要对 SpringBoot 应用的配置约定先做一个简单的介绍。

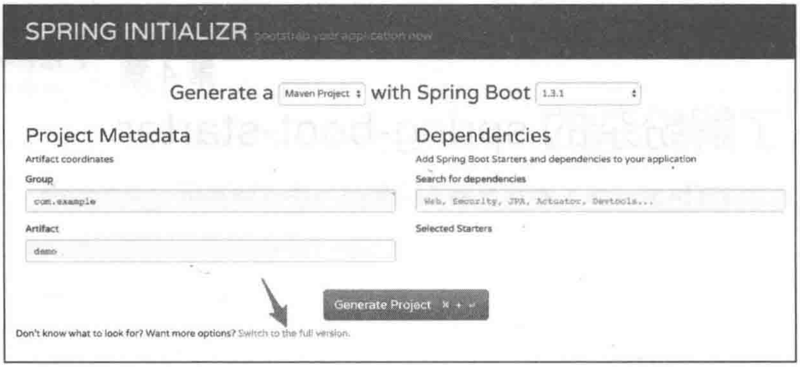


图 4-1　Spring Initializr 示意图

简单来讲，我们可以将对 SpringBoot 的行为可以进行干预的配置方式划分为几类：

- 命令行参数（Command Line Args）。
- 系统环境变量（Environment Variables）。
- 位于文件系统中的配置文件。
- 位于 classpath 中的配置文件。
- 固化到代码中的配置项。

为了简化，其他比较少见场景的配置方式不在这里罗列。总的来说，以上几种方式按照优先级从高到低排列，高优先级方式提供的配置项可以覆盖或者优先生效，比如通过命令行参数传入的配置项会覆盖通过环境变量传入的同一配置项，当然也会覆盖其他后面几种方式给出的同一配置项。

不管是位于文件系统还是 classpath，SpringBoot 应用默认的配置文件名叫作 application.properties，可以直接放在当前项目的根目录下或者名称为 config 的子目录下。

以上是关于 SpringBoot 应用配置方式的简单介绍，基本可以满足我们后面讲解的需要，所以，现在让我们进入纷杂的 spring-boot-starter 探索之旅吧！

4.1　应用日志和 spring-boot-starter-logging

Java 的日志系统多种多样，从 java.util 默认提供的日志支持，到 log4j，log4j2，commons logging 等，复杂繁多，所以，应用日志系统的配置就会比较特殊，从而 spring-boot-starter-logging 也比较特殊一些，下面将其作为我们第一个了解的自动配置依赖模块。

假如 maven 依赖中添加了 spring-boot-starter-logging：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
</dependency>
```

那么，我们的 SpringBoot 应用将自动使用 logback 作为应用日志框架，SpringBoot 启动的时候，由 org.springframework.boot.logging.Logging-Application-Listener 根据情况初始化并使用。

SpringBoot 为我们提供了很多默认的日志配置，所以，只要将 spring-boot-starter-logging 作为依赖加入到当前应用的 classpath，则“开箱即用”，不需要做任何多余的配置，但假设我们要对默认 SpringBoot 提供的应用日志设定做调整，则可以通过几种方式进行配置调整：

- 遵循 logback 的约定，在 classpath 中使用自己定制的 logback.xml 配置文件。
- 在文件系统中任何一个位置提供自己的 logback.xml 配置文件，然后通过 logging.config 配置项指向这个配置文件来启用它，比如在 application.properties 中指定如下的配置。

```
logging.config=/{some.path.you.defined}/any-logfile-name-I-like.log
```

注意　SpringBoot 默认允许我们通过在配置文件或者命令行等方式使用 logging.file 和 logging.path 来自定义日志文件的名称和存放路径，不过，这只是允许我们在 SpringBoot 框架预先定义的默认日志系统设定的基础上做有限的设置，如果我们希望更灵活的配置，最好通过框架特定的配置方式提供相应的配置文件，然后通过 logging.config 来启用。

如果大家更习惯使用log4j或者log4j2，那么也可以采用类似的方式将它们对应的spring-boot-starter依赖模块加到Maven依赖中即可：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j</artifactId>
</dependency>
```

或者

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

但一定不要将这些完成同一目的的spring-boot-starter都加到依赖中。

4.2 快速Web应用开发与spring-boot-starter-web

在这个互联网时代，使用Spring框架除了开发少数的独立应用，大部分情况下实际上在使用SpringMVC开发web应用，为了帮我们简化快速搭建并开发一个Web项目，SpringBoot为我们提供了spring-boot-starter-web自动配置模块。

只要将spring-boot-starter-web加入项目的maven依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

我们就得到了一个直接可执行的Web应用，当前项目下运行mvn spring-boot:run就可以直接启动一个使用了嵌入式tomcat服务请求的Web应用，只不过，我们还没有提供任何服务Web请求的Controller，所以，访问任何路径都会返回一个SpringBoot默认提供的错误页面（一般称其为whitelabel error page），我们可以在当前项目下新建一个服务根路径Web请求的Controller实现：

```
@RestController
public class IndexController {
```

```
    @RequestMapping("/")
    public String index() {
        return "hello, there";
    }
}
```

重新运行 mvn spring-boot:run 并访问 http://localhost:8080，错误页面将被我们的 Controller 返回的消息所替代，一个简单的 Web 应用就这样完成了。

但是，简单的背后，其实却有很多“潜规则”（约定），我们只有充分了解了这些“潜规则”，才能更好地应用 spring-boot-starter-web。

4.2.1 项目结构层面的约定

项目结构层面与传统打包为 war 的 Java Web 应用的差异在于，静态文件和页面模板的存放位置变了，原来是放在 src/main/webapp 目录下的一系列资源，现在都统一放在 src/main/resources 相应子目录下，比如：

1）src/main/resources/static 用于存放各类静态资源，比如 css，js 等。

2）src/main/resources/templates 用于存放模板文件，比如 *.vm。

注意 当然，如果还是希望以 war 包的形式，而不是 SpringBoot 推荐使用的独立 jar 包形式发布 Web 应用，也可以继续原来 Java Web 应用的项目结构约定。

4.2.2 SpringMVC 框架层面的约定和定制

spring-boot-starter-web 默认将为我们自动配置如下一些 SpringMVC 必要组件：

- 必要的 ViewResolver，比如 ContentNegotiatingViewResolver 和 BeanNameViewResolver。
- 将必要的 Converter、GenericConverter 和 Formatter 等 bean 注册到 IoC 容器。
- 添加一系列的 HttpMessageConverter 以便支持对 Web 请求和相应的类型转换。

- ❑ 自动配置和注册 MessageCodesResolver。
- ❑ 其他。

任何时候，如果我们对默认提供的 SpringMVC 组件设定不满意，都可以在 IoC 容器中注册新的同类型的 bean 定义来替换，或者直接提供一个基于 WebMvcConfigurerAdapter 类型的 bean 定义来定制，甚至直接提供一个标注了 @EnableWebMvc 的 @Configuration 配置类完全接管所有 SpringMVC 的相关配置，自己完全重新配置。

4.2.3 嵌入式 Web 容器层面的约定和定制

spring-boot-starter-web 默认使用嵌入式 tomcat 作为 web 容器对外提供 HTTP 服务，默认将使用 8080 端口对外监听和提供服务：

1）假设我们不想使用默认的嵌入式 tomcat（spring-boot-starter-tomcat 自动配置模块），那么可以引入 spring-boot-starter-jetty 或者 spring-boot-starter-undertow 作为替代方案。

2）假设我们不想使用默认的 8080 端口，那么我们可以通过更改配置项 server.port 使用自己指定的端口，比如：

```
server.port=9000
```

spring-boot-starter-web 提供了很多以 server. 为前缀的配置项用于对嵌入式 Web 容器提供配置，比如：

- ❑ server.port
- ❑ server.address
- ❑ server.ssl.*
- ❑ server.tomcat.*

如果这些依然无法满足需求，SpringBoot 甚至允许我们直接对嵌入式的 Web 容器实例进行定制，这可以通过向 IoC 容器中注册一个 EmbeddedServletContainerCustomizer 类型的组件来对嵌入式 Web 容器进行定制：

```
public class UnveilSpringEmbeddedTomcatCustomizer implements Embed-
dedServletContainerCustomizer {
    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
```

```
        container.setPort(9999);
        container.setContextPath("/unveil-spring-chapter3");
        // ...
    }
}
```

再深入的定制则需要针对特定的嵌入式 Web 容器，使用实现对应的 Factory 并注册到 IoC 容器：

- TomcatEmbeddedServletContainerFactory
- JettyEmbeddedServletContainerFactory
- UndertowEmbeddedServletContainerFactory

但是，笔者认为大家几乎没有走到这一步的必要，而且建议最好也不要走到这一步，目前来看，spring-boot-starter-web 提供的配置项列表已经是最简单和完备的定制方式了。

4.3　数据访问与 spring-boot-starter-jdbc

大部分 Java 应用都需要访问数据库，尤其是服务层，所以，SpringBoot 会为我们自动配置相应的数据访问设施。

若想 SpringBoot 为我们自动配置数据访问的基础设施，那么，我们需要直接或者间接地依赖 spring-jdbc，一旦 spring-jdbc 位于我们 SpringBoot 应用的 classpath，即会触发数据访问相关的自动配置行为，最简单的做法就是把 spring-boot-starter-jdbc 加为应用的依赖。

默认情况下，如果我们没有配置任何 DataSource，那么，SpringBoot 会为我们自动配置一个基于嵌入式数据库的 DataSource，这种自动配置行为其实很适合于测试场景，但对实际的开发帮助不大，基本上我们会自己配置一个 DataSource 实例，或者通过自动配置模块提供的配置参数对 DataSource 实例进行自定义的配置。

假设我们的 SpringBoot 应用只依赖一个数据库，那么，使用 DataSource 自动配置模块提供的配置参数是最方便的：

```
spring.datasource.url=jdbc:mysql://{database host}:3306/{databaseName}
spring.datasource.username={database username}
```

```
spring.datasource.password={database password}
```

当然，自己配置一个 DataSource 也是可以的，SpringBoot 也会智能地选择我们自己配置的这个 DataSource 实例（只不过必要性真不大）。

除了 DataSource 会自动配置，SpringBoot 还会自动配置相应的 JdbcTemplate、DataSourceTransactionManager 等关联“设施”，可谓服务周到，我们只要在使用的地方注入就可以了：

```
class SomeDao {
    @Autowired
    JdbcTemplate jdbcTemplate;

    public <T> List<T> queryForList(String sql){
        // ...
    }

    // ...
}
```

不过，spring-boot-starter-jdbc 以及与其相关的自动配置也不总是带来便利，在某些场景下，我们可能会在一个应用中需要依赖和访问多个数据库，这个时候就会出现问题了。

假设我们在 ApplicationContext 中配置了多个 DataSource 实例指向多个数据库：

```
@Bean
public DataSource dataSource1() throws Throwable {
    DruidDataSource dataSource = new DruidDataSource();
    dataSource.setUrl(...);
    dataSource.setUsername(...);
    dataSource.setPassword(...);
    // TODO other settings if necessary in the future.
    return dataSource;
}

@Bean
public DataSource dataSource2() throws Throwable {
    DruidDataSource dataSource = new DruidDataSource();
    dataSource.setUrl(...);
    dataSource.setUsername(...);
    dataSource.setPassword(...);
    // TODO other settings if necessary in the future.
```

```
    return dataSource;
}
```

那么，不好意思，启动SpringBoot应用的时候会抛出类似如下的异常（Exception）：

```
No qualifying bean of type [javax.sql.DataSource] is defined:
expected single matching bean but found 2
```

为了避免这种情况的发生，我们需要在SpringBoot的启动类上做点儿“手脚”：

```
@SpringBootApplication(exclude = {
        DataSourceAutoConfiguration.class,
        DataSourceTransactionManagerAutoConfiguration.class
})
public class UnveilSpringChapter3Application {

    public static void main(String[] args) {
        SpringApplication.run(UnveilSpringChapter3Application.
class, args);
    }
}
```

也就是说，我们需要在这种场景下排除掉对SpringBoot默认提供的DataSource相关的自动配置。

但如果我们还是想要享受SpringBoot提供的自动配置DataSource的机能，也可以通过为其中一个DataSource配置添加org.springframework.context.annotation.Primary这个Annotation的方式以实现两全其美：

```
@Bean
@Primary
public DataSource dataSource1() throws Throwable {
    DruidDataSource dataSource = new DruidDataSource();
    dataSource.setUrl(...);
    dataSource.setUsername(...);
    dataSource.setPassword(...);
    // TODO other settings if necessary in the future.
    return dataSource;
}

@Bean
public DataSource dataSource2() throws Throwable {
```

```
    DruidDataSource dataSource = new DruidDataSource();
    dataSource.setUrl(...);
    dataSource.setUsername(...);
    dataSource.setPassword(...);
    // TODO other settings if necessary in the future.
    return dataSource;
}
```

另外，SpringBoot 还提供了很多其他数据访问相关的自动配置模块，比如 spring-boot-starter-data-jpa、spring-boot-starter-data-mongodb 等，大家可以根据自己数据访问的具体场景选择使用这些自动配置模块。

警告 如果选择了 spring-boot-starter-data-jpa 等关系数据库相关的数据访问自动配置模块，并且还需要同时依赖访问多个数据库，那么，也需要相应的在 SpringBoot 启动类中排除掉这些自动配置模块中的 AutoConfiguration 实现类（对应 spring-boot-starter-data-jpa 是 JpaRepositoriesAutoConfiguration），或者标注某个 DataSource 为 @Primary。

4.3.1 SpringBoot 应用的数据库版本化管理

关于如何针对数据库的变更进行版本化管理，从 Ruby On Rails 的 migration 支持，到 Java 的 MyBatis Migrations，Flyway 以及 Liquibase，都给出了相应的最佳实践建议和方案，但是，从我所看到的国内业界现状，数据库 migrations 的实践方式并没有在国内普遍应用起来，大部分都是靠人来解决，这或许可以用一句“成熟度不够”来解释，另外一个原因或许是职能明确分工后造成的局面。

如果仔细分析以上数据库 migration 方案就会发现，它们给出的应用场景和实践几乎都是单应用、单部署的，这在庞大单一部署单元（Monolith）的年代显然是很适合的，因为应用从开发到发布部署，再到启动，整个生命周期内，应用相关的所有“原材料”都集中在一起进行管理，而且国外开发者往往偏“特种作战”（Full-Stack Developer），一身多能，从而数据库 migration 这种实践自然可以成型并广泛应用。

但回到国内来看，我们往往是“集团军作战”，拼的是“大部队 + 明确分工”的模式，而且应用所面向的服务人数也往往更为庞大，所以，整个应用的

交付链路上各个环节之间的衔接是不同的人，而应用最终部署的拓扑又往往是分布式部署居多，所以，在一个项目单元里维护数据库的 migration 脚本然后部署后启动前执行这些脚本就变得不合时宜了：

1）从职责上，这些 migration 脚本虽然大部分情况下都是开发人员写，但写完之后要不要进行 SQL 审查，是否符合规范，这些又会涉及应用运维 DBA，代码管理系统对开发来说很亲切，对 DBA 来说则不尽然，而且 DBA 往往还要一人服务多个团队多个项目，从 DBA 的角度来说，他更愿意将 SQL 集中到一处进行管理，而不是分散在各个项目中。

2）应用分布式部署之后，就不单单是单一部署在应用启动的之前直接执行一次 migration 脚本那么简单了，你要执行多次，虽然 migration 方案都有版本控制，变更应该最终状态都是一样的，但这多个部署节点上都执行同一逻辑显然是多余的。更复杂一点儿，多个应用可能同时使用同一个数据库的情况（不要怀疑，遗留系统对大家来说并不陌生），一个项目的数据库 migration 操作跟另一个项目的数据库 migration 操作会不会在互不知晓的情况下产生冲突和破坏？

所以，数据库 migration 的思路和实践很好，但不能照搬（任何事情其实皆如此），不过，我们倒是不用一棒子打死，结合现有的一些数据库 migration 方案，比如 flyway 或者 liquibase，我们可以对这些数据库 migration 的基础设施和支持外部化（Externalize），一个可能的架构如图 4-2 所示。

在这个架构中，数据库 migration 的版本化管理剥离到了单独的管理系统，单一项目中不再保存完整历史的 migration 记录，而只需要提供当次发布要牵扯的数据库变更 SQL。在项目发布的时候，由 DBA 进行统一的审查并纳入单独的数据库 migration 管理系统，由单独的数据库 migration 管理系统来管理完整的数据库 migration 记录，可以根据数据库的粒度进行管理和状态同步，从而既可以在开发阶段让开发人员可以集中管理数据库 SQL，又能在发布期间审查 SQL 并同步 migration 状态和完整的历史记录管理。当然，这一切可以实现的前提是有一套完整的软件交付链路支撑平台，能够从流程上，软件生命周期管理上进行统一的治理和规范，此为后话，我们将在下一章跟大家做进一步深入的探讨。

不管怎么样，SpringBoot 还是为大家提供了针对 Flyway 和 Liquibase 的自动

配置功能（org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration 和 org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration），对于单一开发和部署的应用来说，还是可以考虑的。

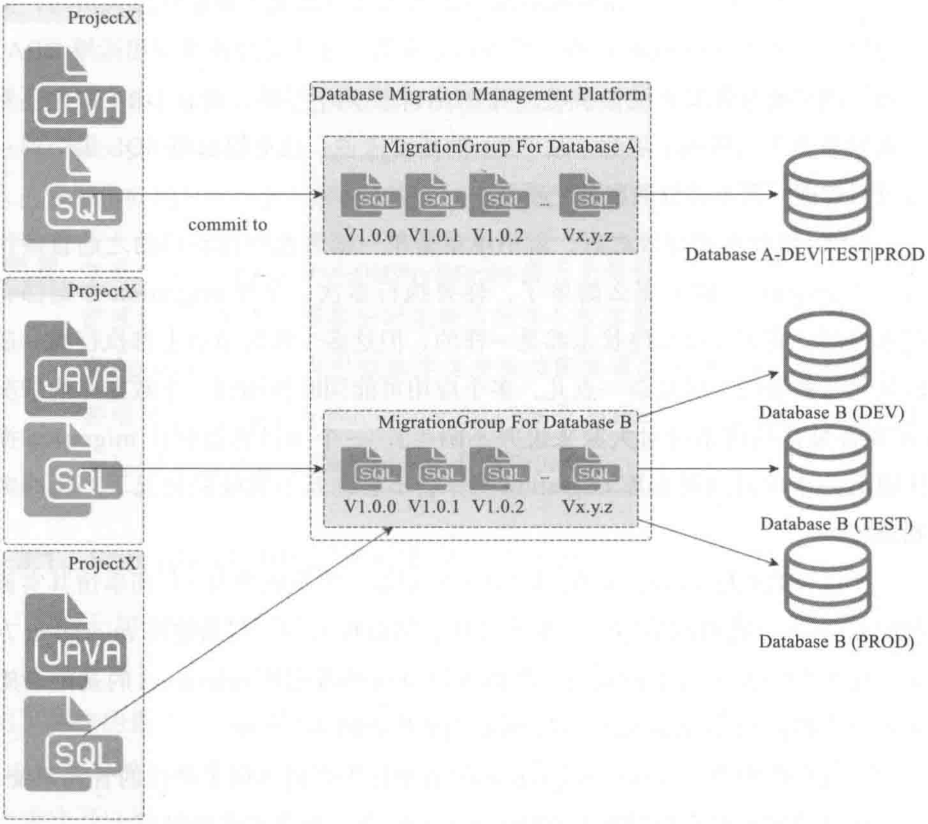


图 4-2　集中管控的数据库 Migration 架构示意图

4.4　spring-boot-starter-aop 及其使用场景说明

如今，AOP（Aspect Oriented Programming）已经不是什么崭新的概念了，在经历了代码生成、动态代理、字节码增强甚至静态编译等不同时代的洗礼之后，Java 平台上的 AOP 方案基本上已经以 SpringAOP 结合 AspectJ 的方式稳固下来（虽然大家依然可以自己通过各种字节码工具偶尔“打造一些轮子”）。

现在 Spring 框架提供的 AOP 方案倡导了一种各取所长的方案，即使

用 SpringAOP 的面向对象的方式来编写和组织织入逻辑，并使用 AspectJ 的 Pointcut 描述语言配合 Annotation 来标注和指明织入点（Jointpoint）。

原则上来说，我们只要引入 Spring 框架中 AOP 的相应依赖就可以直接使用 Spring 的 AOP 支持了，不过，为了进一步为大家使用 SpringAOP 提供便利，SpringBoot 还是“不厌其烦”地为我们提供了一个 spring-boot-starter-aop 自动配置模块。

spring-boot-starter-aop 自动配置行为由两部分内容组成：

1）位于 spring-boot-autoconfigure 的 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration 提供 @Configuration 配置类和相应的配置项。

2）spring-boot-starter-aop 模块自身提供了针对 spring-aop、aspectjrt 和 aspectjweaver 的依赖。

一般情况下，只要项目依赖中加入了 spring-boot-starter-aop，其实就会自动触发 AOP 的关联行为，包括构建相应的 AutoProxyCreator，将横切关注点织入（Weave）相应的目标对象等，不过 AopAutoConfiguration 依然为我们提供了可怜的两个配置项，用来有限地干预 AOP 相关配置：

- spring.aop.auto=true
- spring.aop.proxy-target-class=false

对我们来说，这两个配置项的最大意义在于：允许我们投反对票，比如可以选择关闭自动的 aop 配置（spring.aop.auto=false），或者启用针对 class 而不是 interface 级别的 aop 代理（aop proxy）。

AOP 的应用场景很多，我们不妨以当下最热门的 APM（Application Performance Monitoring）为实例场景，尝试使用 spring-boot-starter-aop 的支持打造一个应用性能监控的工具原型。

4.4.1 spring-boot-starter-aop 在构建 spring-boot-starter-metrics 自定义模块中的应用

对于应用性能监控来说，架构逻辑上其实很简单，基本上就是三步走（如图 4-3 所示）。

本节暂时只构建一个 spring-boot-starter-metrics 自定义的自动配置模块用来解决“应用性能数据采集”的问题。

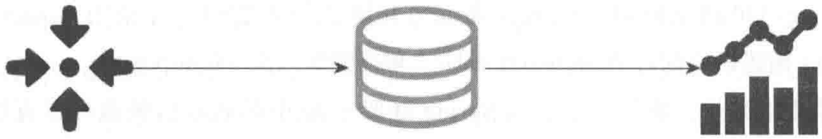


图 4-3 应用性能监控关键环节示意图

在此之前，有几个原则我们需要先说明一下：

- 虽然说采集应用性能数据可以帮助我们更好地分析和改进应用的性能指标，但这不意味着可以借着 APM 的名义对应用的核心职能形成侵害，加上应用性能数据采集功能一定会对应用的性能本身带来拖累，你拿到的所谓性能数据是分摊了你的数据采集方案带来的负担，所以，一般情况下，最好把应用性能数据采集模块的性能损耗控制在 10% 以内甚至更小。
- SpringAOP 其实提供了多种横切逻辑织入机制（Weaving），性能损耗上也是各有差别，从运行期间的动态代理和字节码增强 Weavng，到类加载期间的 Weaving，甚至高冷的 AspectJ 二次静态编译 Weaving，大家可以根据情况灵活把握。
- 针对应用性能数据的采集，最好对应用开发者是透明的，通过配置外部化的形式，可以最大限度地剥离这部分对应用开发者来说非核心的关注点，只在部署和运行之前确定采集点并配置上线即可，虽然本节实例采用基于 @Annotation 的方式来标注性能采集点，但不意味着这是最优的方式，更多是基于技术方案（SpringAOP）的现状给出的一种实践方式。

下面我们正式着手构建 spring-boot-starter-metrics 自定义的自动配置模块的设计和实现方案。

笔者一向是只在有必要的时候才重新“造轮子”，绝不会为了炫技而去“造轮子”，所以，本次的主角我们选择 Java 中的 Dropwizard Metrics 这个类库作为打造我们 APM 原型的起点。

Dropwizard Metrics 为我们提供了多种不同类型的应用数据度量方案，且通过相应的数据处理算法在性能和批量状态的管理上做了很优秀的工作，只不过，如果我们直接用它的 API 来对自己的应用代码进行度量的话，那写起来代码太多，而且这些性能代码混杂在应用的核心逻辑执行路径上，一个是界面不

友好，另外一个就是不容易维护：

```
public class MockService implements InitializingBean{

    @Autowired
    MetricRegistry metricRegistry;

    private Timer timer;
    private Counter counter;
    // define more other metrics...

    public void doSth(){
        counter.inc();

        Timer.Context context = timer.time();
        try{
            System.out.println("just do something.");
        }finally {
            context.stop();
        }
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        timer = metricRegistry.timer("timerToProfilingDoSthMethod");
        counter = metricRegistry.counter("counterForDoSthMethod");
    }
}
```

所以，对于这些非功能性的性能度量代码，我们可以使用 AOP 的方式剥离到相应的 Aspect 中单独维护，而为了能够将这些性能度量的 Aspect 挂接到指定的待度量代码上，基于现有的方案选型，可以使用 metrics-annotation 提供的一系列 Annotation 来标注织入位置，这样，开发者只要在需要度量的代码位置上标注相应的 Annotation，我们提供的 spring-boot-starter-metrics 自定义的自动配置模块就会自动地收集这些位置上指定的性能度量数据。

首先，我们通过 http://start.spring.io/ 构建一个 SpringBoot 的脚手架项目[⊖]，

⊖ http://start.spring.io/ 部署的是一个 SPRING INITIALIZR 项目，开放源码于 https://github.com/spring-io/initializr，如果因为某些未知原因无法访问 http://start.spring.io/，可以考虑通过 git clone git@github.com:spring-io/initializr.git 到本地、自己或者公司内部搭建一套同样的本地环境，当然，某个模板里面因为引用了某个公司的 css 资源，因未知原因可能导致页面加载缓慢，请移除相应的 css 资源引用重启即可。

选择以 Maven 编译（选择用 Gradle 的同学自行甄别后面的配置如何具体进行），然后在创建好的 SpringBoot 脚手架项目的 pom.xml 中添加如下必要配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol</groupId>
    <artifactId>spring-boot-starter-metrics</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-starter-metrics</name>
    <description>auto configuration module for dropwizard metrics</
description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.0.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>1.8</java.version>
        <metrics.version>3.1.2</metrics.version>
    </properties>

    <!-- 其他配置 -->

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
        <dependency>
```

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>io.dropwizard.metrics</groupId>
            <artifactId>metrics-core</artifactId>
            <version>${metrics.version}</version>
        </dependency>
        <dependency>
            <groupId>io.dropwizard.metrics</groupId>
            <artifactId>metrics-annotation</artifactId>
            <version>${metrics.version}</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.8.7</version>
        </dependency>
    </dependencies>

</project>
```

pom.xml 中有几个关键配置需要关注：

- 继承了 spring-boot-starter-parent，用于加入 springboot 的相关依赖。
- 添加了 spring-boot-starter-aop 依赖，目的你懂的，本章节的主角就是它。
- 添加了 io.dropwizard.metrics 下相应的依赖，用来引入 dropwizard metrics 类库和必要的 Annotations。
- 添加了 spring-boot-starter-actuator，这个自动配置模块我们会在下章跟大家进一步介绍，在这里我们主要是引入它对 dropwizard metrics 和 JMX 的一部分自动配置逻辑，比如针对 MetricRegistry 和 MBeanServer 的自动配置，这样我们就可以直接 @Autowired 来注入使用 MetricRegistry 和 MBeanServer。
- 至于 aspectjrt，是使用了最新的版本，原则上 spring-boot-starter-aop 已经有依赖，这里可以不用明确添加配置。

如果单单是一个提供必要依赖的自动配置模块，那么到这里其实就可以结束了，但我们的 spring-boot-starter-metrics 需要使用 AOP 提供相应的横切关注

点逻辑，所以，还需要编写并提供一些必要的代码组件，因此，最少我们先要提供一个 @Configuration 配置类，用于将我们即将提供的这些 AOP 逻辑暴露给使用者：

```
@Configuration
@ComponentScan({"com.keevol.springboot.metrics.lifecycle", "com.
keevol.springboot.metrics.aop"})
@AutoConfigureAfter(AopAutoConfiguration.class)
public class DropwizardMetricsMBeansAutoConfiguration {

    @Value("${metrics.mbeans.domain.name:com.keevol.metrics}")
    String metricsMBeansDomainName;

    @Autowired
    MBeanServer mbeanServer;

    @Autowired
    MetricRegistry metricRegistry;

    @Bean
    public JmxReporter jmxReporter() {
        JmxReporter reporter = JmxReporter
                .forRegistry(metricRegistry)
                .inDomain(metricsMBeansDomainName)
                .registerWith(mbeanServer)
                .build();
        return reporter;
    }
}
```

然后就是将这个配置类添加到 META-INF/spring.factories：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.keevol.springboot.metrics.autocfg.DropwizardMetricsMBeansAuto-
ConfigurationOK,
```

不要认为将 spring-boot-starter-metrics 打包作为类库发布出去就可以了，AOP 相关的代码还没写。

我们回头来看 DropwizardMetricsMBeansAutoConfiguration 配置类，这个配置类的实现很简单，注入了 MBeanServer 和 MetricRegistry 的实例，并开放了一个 metrics.mbeans.domain.name 配置属性（默认值 com.keevol.metrics）便于使用者指定自定义的 MBean 暴露和访问的命名空间，当然，这些其实都不是

重点，因为它们都只是为了将我们要采集的性能数据指标以 JMX 的形式暴露出去而服务的，重点在于 DropwizardMetricsMBeansAutoConfiguration 头顶上的那几顶“帽子”：

- @Configuration 自然不必说了，这是一个 JavaConfig 配置类。
- @ComponentScan({"com.keevol.springboot.metrics.lifecycle", "com.keevol.springboot.metrics.aop"})，为了简便，让 @ComponentScan 把这两个 java package 下的所有组件都加载到 IoC 容器中，这些组件就包括我们要提供的一系列与 AOP 和 Dropwizard Metrics 相关的实现逻辑。
- @AutoConfigureAfter(AopAutoConfiguration.class) 告诉 SpringBoot：“我希望 DropwizardMetricsMBeansAutoConfiguration 在 AopAutoConfiguration 完成之后进行配置”。

现在，最后的秘密就隐藏在 @ComponentScan 背后的两个 java package 之下了。

首先是 com.keevol.springboot.metrics.aop，在这个 java package 下面，我们只提供了一个 AutoMetricsAspect，其定义如下：

```
@Component
@Aspect
public class AutoMetricsAspect {

    protected ConcurrentMap<String, Meter> meters = new
ConcurrentHashMap<>();
    protected ConcurrentMap<String, Meter> exceptionMeters = new
ConcurrentHashMap<>();
    protected ConcurrentMap<String, Timer> timers = new
ConcurrentHashMap<>();
    protected ConcurrentMap<String, Counter> counters = new
ConcurrentHashMap<>();

    @Autowired
    MetricRegistry metricRegistry;

    @Pointcut(value = "execution(public * *(..))")
    public void publicMethods() {
    }

    @Before("publicMethods() && @annotation(countedAnnotation)")
    public void instrumentCounted(JoinPoint jp, Counted
```

```
countedAnnotation) {
        String name = name(jp.getTarget().getClass(), StringUtils.
hasLength(countedAnnotation.name()) ? countedAnnotation.name() :
jp.getSignature().getName(), "counter");
        Counter counter = counters.computeIfAbsent(name, key ->
metricRegistry.counter(key));
        counter.inc();
    }

    @Before("publicMethods() && @annotation(meteredAnnotation)")
    public void instrumentMetered(JoinPoint jp, Metered
meteredAnnotation) {
        String name = name(jp.getTarget().getClass(), StringUtils.
hasLength(meteredAnnotation.name()) ? meteredAnnotation.name() :
jp.getSignature().getName(), "meter");
        Meter meter = meters.computeIfAbsent(name, key ->
metricRegistry.meter(key));
        meter.mark();
    }

    @AfterThrowing(pointcut = "publicMethods() && @annotation(exMe-
teredAnnotation)", throwing = "ex")
    public void instrumentExceptionMetered(JoinPoint jp, Throwable
ex, ExceptionMetered exMeteredAnnotation) {
        String name = name(jp.getTarget().getClass(), StringUtils.
hasLength(exMeteredAnnotation.name()) ? exMeteredAnnotation.name() :
jp.getSignature().getName(), "meter", "exception");
        Meter meter = exceptionMeters.computeIfAbsent(name,
meterName -> metricRegistry.meter(meterName));
        meter.mark();
    }

    @Around("publicMethods() && @annotation(timedAnnotation)")
    public Object instrumentTimed(ProceedingJoinPoint pjp, Timed
timedAnnotation) throws Throwable {
        String name = name(pjp.getTarget().getClass(), StringUtils.
hasLength(timedAnnotation.name()) ? timedAnnotation.name() : pjp.
getSignature().getName(), "timer");
        Timer timer = timers.computeIfAbsent(name, inputName ->
metricRegistry.timer(inputName));
        Timer.Context tc = timer.time();
        try {
            return pjp.proceed();
        } finally {
            tc.stop();
```

```
            }
        }

    }
```

@Aspect + @Component 的目的在于告诉 Spring 框架：“我是一个 AOP 的 Aspect 实现类并且你可以通过 @ComponentScan 把我加入 IoC 容器之中。”当然，这不是重点。

io.dropwizard.metrics:metrics-annotation 这个依赖包为我们提供了几个有趣的 Annotation：

- ❑ Timed
- ❑ Gauge
- ❑ Counted
- ❑ Metered
- ❑ ExceptionMetered

这些语义良好的 Annotation 定义可以用来标注相应的 AOP 逻辑扩展点，比如，针对同一个 MockService，我们可以将性能数据的度量和采集简化为只标注一两个 Annotation 就可以了：

```
@Component
public class MockService {

    @Timed
    @Counted
    public void doSth() {
        System.out.println("just do something.");
    }
}
```

但是，Annotation 注定只是 Annotation，它们只是一些标记信息，要让它们发挥作用，需要有“伯乐”的眷顾，所以，AutoMetricsAspect 在这里就是这些 Dropwizard Metrics Annotation 的“伯乐”，通过拦截每一个 public 方法并检查方法上是否存在某个 metrics annotation，我们就可以根据具体的 metrics annotation 的类型，为匹配的方法注入相应性能数据采集代码逻辑，从而完成整个基于 AOP 和 dropwizard metrics 的应用性能数据采集方案的实现。

注意 受限于 SpringAOP 自身的一些限制，并不是所有 AOP 的 Joinpoint 类型都支持，而且，以上原型代码方向也不见得是性能最优的方案，大家需要结合自己的目标和手上可用的技术手段，根据自己的具体应用场景具体分析和权衡，这里权且做抛砖引玉。

至此，整个基于 spring-boot-starter-aop 的 spring-boot-starter-metrics 自定义自动配置模块宣告完工，不知道对各位是否有所启发？

注意 对于想了解更多细节，或者想寻找直接可用方案的读者，可以参考开源项目 https://github.com/ryantenney/metrics-spring。

4.5 应用安全与 spring-boot-starter-security

应用安全属于安全防护体系中的重要一环，但也是最薄弱的一环（我个人认为），究其原因，或许是：

- 应用的核心职责是完成业务和产品的功能需求，而安全确实非功能性需求，在资源有限的情况下，企业一定是更加注重将有限的资源投入到“开疆扩土”上去，否则，穷家破瓦的，也真没有什么值得安全防护的。
- 大部分应用开发者对应用安全知之甚少，而且安全一般属于一个企业或者业界秘而不宣的信息，所以，在没有一个专职的安全团队负责推动整个安全防护体系落实的情况下，零零散散和线上落实的一些应用安全防护已经算很不错的了。
- “树大招风”，树不大的时候，那些“风”通常也不会来找你麻烦，所以，大部分中小企业，除非应用了一些业界广泛应用的软件或者方案被连带性地伤害到，大部分情况下，这些中小企业并不知道这类潜在风险，或者是他们自身不会为黑客们（cracker）带来太大的利用价值。

好在我们这些 Java 开发者生活在 Spring 框架营造的生态圈之中，所以，关于应用安全这种头疼的问题，Spring 生态圈里也有现成的解决方案，即从 Acegi 发展起来的 SpringSecurity。但是说实话，SpringSecurity 在整个社区中的名声并不是太好，尤其是在开发者眼中，“复杂（Too Complicated），太

重（Too Heavyweight）"，但实际上，如果大家真得扑进去了解一个人，哦，不对，一个框架，就会发现，其实 SpringSecurity 框架本身的设计还是挺优秀的，SpringSecurity 可以任意裁剪，而且还提供了丰富的开箱即用的安全特性支持。这里其实存在一个常见的设计取舍，我们到底应该为了良好的扩展和组合型而将组件拆分的精细一些，还是应该为了使用的便利，适度忽略定制化的需求，提供一个功能简化的一站式方案？不管 SpringSecurity 团队当时是如何选择的，既然已经成为了事实，给使用者的感受不好，那么，我们就要想办法改善这种现状，spring-boot-starter-security 就是一种答案。

spring-boot-starter-security 主要面向 Web 应用安全，配合 spring-boot-starter-web，要使用 SpringBoot 构建一个安全的对外提供服务的 Web 应用简直太容易了：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.unveilspring.chapter3</groupId>
    <artifactId>web-security-demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>web-security-demo</name>
    <description>web security demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.1.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
```

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- 其他依赖 -->
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

在当前项目中只要添加需要的 Controller 实现，一个添加了基本安全防护的 Web 应用就诞生了。spring-boot-starter-security 默认会提供一个基于 HTTP Basic 认证的安全防护策略，默认用户名为 user，访问密码则在当前 Web 应用启动的时候，打印到控制台，类似于：

```
2016-01-01 13:57:00.596  INFO 17966 --- [ost-startStop-1] b.a.s.Au-
thenticationManagerConfiguration :

Using default security password: 560ff91b-0ae7-492c-ad16-603e1adec54c
```

如果我们希望对 HTTP Basic 认证的用户名和密码进行定制，可以通过如下配置项进行：

```
security.user.name={ 个人希望设置的用户名 }
security.user.password={ 个人希望使用的访问密码 }
```

除此之外，spring-boot-starter-security 还会默认启用一些必要的 Web 安全防护，比如针对 XSS、CSRF 等常见针对 Web 应用的攻击，同时，也会将一些常见的静态资源路径排除在安全防护之外。

但是，说实话，spring-boot-starter-security 提供的默认安全策略相对于真正的生产环境来说，还是太弱了，但也没办法，既要安全，又要便利，spring-

boot-starter-security 默认情况下已经尽量做到够好了。不过好在 SpringSecurity 扩展性不错，要在其上构建一套真正严谨有效的 Web 应用安全防护体系也并非难事，只不过，需要我们先能够从其架构设计上理解并把握它，然后再在 SpringSecurity 和 SpringBoot 的基础上构建一套符合自身需要的 Web 应用安全方案。

4.5.1　了解 SpringSecurity 基本设计

SpringSecurity 框架不但囊括了基本的认证和授权功能，而且还提供了加密解密、统一登录等一系列相关支持，毕竟不是一本专讲 SpringSecurity 的书，所以，本节中只是对框架比较核心的设计进行简单介绍，即基本的认证和授权为核心的设计。

我们可以将 Spring Security 的几个核心概念按照图 4-4 所示勾勒在一起：

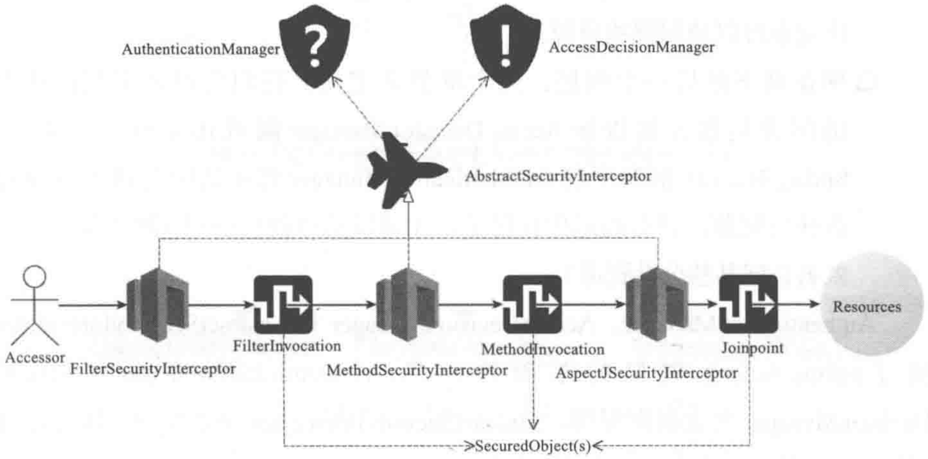


图 4-4　SpringSecurity 核心概念示意图

- 访问者（Accessor）需要访问某个资源（Resources）是这个场景中最原始的需求，但并不是谁都可以访问资源，也不是任何资源都允许任何人来访问，所以，中间我们要加入一些检查和防护。
- 在访问资源的所经之路上，可能需要上山、过桥、下海行船，不管怎么样，这些所经之路对于我们要防护的资源来说都是比较好的设置关卡点，对应上图就是 FilterInvocation（对应 Web 应用场景）、

MethodInvocation 以及 Joinpoint，这些在 Spring Security 框架中统称 Secured Object(s)。

- 我们知道了在哪里设置关卡最合适，下一步就是设置关卡，对应不同的所经之路，我们分别设置类似 FilterSecurityInterceptor、Method-SecurityInterceptor 以及 AspectJSecurityInterceptor 这样的关卡来负责拦截非法资源访问的闯入者们；而在 Spring Security 框架的设计中，关卡的概念统一抽象为 AbstractSecurityInterceptor，而 FilterSecurity-Interceptor、MethodSecurityInterceptor 以及 AspectJSecurityInterceptor 都是它的具体实现类。
- 好啦，把门儿的倒是有了，可是他们不知道该拦谁，不该拦谁，所以，我们需要有类似神盾局（S.H.I.E.L.D）这样的机构，由这个机构来决定谁可以放行，谁必须阻截，而在 SpringSecurity 框架中 AccessDecisionManager 就是这个控制机构，AccessDecisionManager 将决定谁可以访问哪些资源。
- 现在剩下最后一个问题，这个谁怎么定义？我们总得知道当前这个访问者是谁才能告知 AccessDecisionManager 阻截还是放行，所以，SpringSecurity 框架中的 AuthenticationManager 将解决的是访问者身份认证的问题，只有确定你在册了，才可以给你授权访问（除非我们运行匿名访问某些公共资源）。

AuthenticationManager、AccessDecisionManager 和 AbstractSecurityInterceptor 属于 Spring Security 框架的基础铁三角，AuthenticationManager 和 Access-DecisionManager 负责制定规则，AbstractSecurityInterceptor 负责执行。所有针对不同应用场景的安全方案，基本上都是在这个基础核心的基础上衍生出来的，比如，Web 安全。

Spring Security 的 Web 安全方案基于 Java 的 Servlet API 规范进行构建，所以，像 Play Framework 这种脱离 Servlet 规范的 Web 框架，则无法享受到 SpringSecurity 提供的默认的 Web 安全方案（当然，依然可以基于基本模型实现扩展方案）。

既然是基于 Servlet API 规范，那么，要实现关卡的“特效”，则非 javax.servlet.Filter 莫属了。在使用 Spring 框架开发 Filter 的时候，为了让 Filter

可以享受到依赖注入的好处，我们一般是实现 GenericFilterBean 并注册到 IoC 容器，为了能够启用这些注册到 IoC 容器的 Filter，我们一般要在 web.xml 或者相应的 JavaConfig 的配置中声明一个 org.springframework.web.filter.DelegatingFilterProxy，使其 filter-name 与 IoC 容器中我们希望启用的 Filter 对应“挂钩”，SpringSecurity 的 Web 安全方案的启用也是这个原理。SpringSecurity 默认会需要声明一个默认名称为“springSecurityFilterChain”的 org.springframework.web.filter.DelegatingFilterProxy（web.xml 方式或者 JavaConfig 方式），然后指向 IoC 容器中注册的一个 org.springframework.security.web.FilterChainProxy 实例，FilterChainProxy 通过扩展 GenericFilterBean 间接实现了 Filter 接口，同时持有一组 SecurityFilterChain，使它可以针对不同的 Web 资源进行特定的防护，这些“角儿”之间的关系大体上如图 4-5 所示。

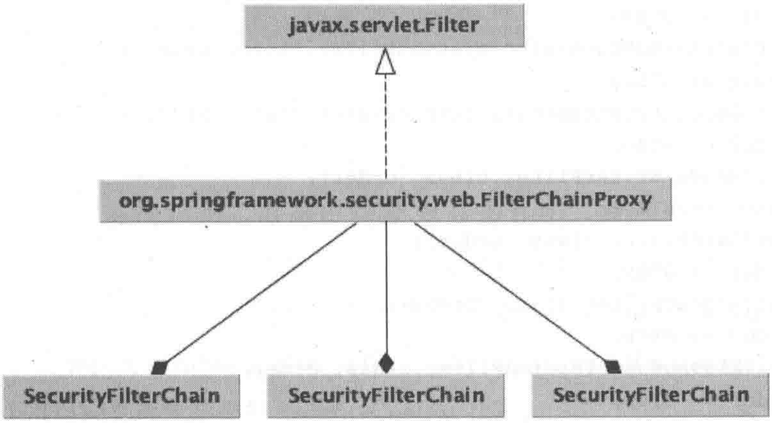


图 4-5　FilterChainProxy 相关组件关系示意图

当然，这些还只是“骨架”，真正执行防护任务的其实是一个个 org.springframework.security.web.SecurityFilterChain 中定义的一系列 Filter：

```
public interface SecurityFilterChain {
    boolean matches(HttpServletRequest request);
    List<Filter> getFilters();
}
```

当我们经常看到如下的 xml schema 形式的配置格式的时候：

```
<http auto-config='true'>
```

```
    <intercept-url pattern="/login.jsp*" access="IS_AUTHENTICATED_
  ANONYMOUSLY"/>
    <intercept-url pattern="/**" access="ROLE_USER" />
    <form-login login-page='/login.jsp'/>
  </http>
```

其实一个个 http 元素背后对应的就是一个个 SecurityFilterChain 实例，而 http 元素的那些子元素，比如 intercept-url，则对应的就是一个个 Filter。

默认情况下，Spring Security 为 SecurityFilterChain 中的 Filter 序列设定了一个注册框架，以 100 为间隔步长，按照一个合理的顺序来规划和排布常用的 Filter 实现（代码参考 FilterComparator）：

```
int order = 100;
put(ChannelProcessingFilter.class, order);
order += STEP;
put(ConcurrentSessionFilter.class, order);
order += STEP;
put(WebAsyncManagerIntegrationFilter.class, order);
order += STEP;
put(SecurityContextPersistenceFilter.class, order);
order += STEP;
put(HeaderWriterFilter.class, order);
order += STEP;
put(CsrfFilter.class, order);
order += STEP;
put(LogoutFilter.class, order);
order += STEP;
put(X509AuthenticationFilter.class, order);
order += STEP;
put(AbstractPreAuthenticatedProcessingFilter.class, order);
order += STEP;
filterToOrder.put("org.springframework.security.cas.web.
CasAuthenticationFilter", order);
order += STEP;
put(UsernamePasswordAuthenticationFilter.class, order);
order += STEP;
put(ConcurrentSessionFilter.class, order);
order += STEP;
filterToOrder.put("org.springframework.security.openid.
OpenIDAuthenticationFilter", order);
order += STEP;
put(DefaultLoginPageGeneratingFilter.class, order);
order += STEP;
```

```
put(ConcurrentSessionFilter.class, order);
order += STEP;
put(DigestAuthenticationFilter.class, order);
order += STEP;
put(BasicAuthenticationFilter.class, order);
order += STEP;
put(RequestCacheAwareFilter.class, order);
order += STEP;
put(SecurityContextHolderAwareRequestFilter.class, order);
order += STEP;
put(JaasApiIntegrationFilter.class, order);
order += STEP;
put(RememberMeAuthenticationFilter.class, order);
order += STEP;
put(AnonymousAuthenticationFilter.class, order);
order += STEP;
put(SessionManagementFilter.class, order);
order += STEP;
put(ExceptionTranslationFilter.class, order);
order += STEP;
put(FilterSecurityInterceptor.class, order);
order += STEP;
put(SwitchUserFilter.class, order);
```

这些Filter虽然很多，但可以简单划分为几类，除个别Filter在每个SecurityFilterChain都需要，其他可以根据需要选用并添加：

- 第一类可以认为是信道与状态管理，比如ChannelProcessingFilter用于处理http或者https之间的切换，而SecurityContextPersistenceFilter用于重建或者销毁必要的SecurityContext状态。
- 第二类是常见Web安全防护类，比如CsrfFilter。
- 第三类是认证和授权，比如BasicAuthenticationFilter、CasAuthenticationFilter等。

最需要重点关注的是FilterSecurityInterceptor，还记得我们前面说到的secured object吧。FilterSecurityInterceptor就属于放在Web访问路径上的那道“关卡”，现在，它的真实位置和效能终于浮出水面了。

ExceptionTranslationFilter属于另一个需要关注的核心类，它负责接待或者送客，如果访客来访，对方没有报上名来，那么，它会让访客去登记认证（去找AuthenticationManager做认证）；如果对方报上名了，但认证失败，那

么不好意思，请重新认证或者走人。当然，它拒绝访客的方式是抛出相应的 Exception，所以名字叫 ExceptionTranslationFilter。

最后，这个 Filter 序列因为间隔了 100 的步长，所以，我们可以在其中穿插自己的 Filter 实现类，为定制和扩展 SpringSecurity 的防护体系提供了机会。

以上就是关于 Spring Security 以及其 Web 安全相关的基础介绍，这些内容足以帮助我们理解并扩展 spring-boot-starter-security。

提示 如果大家还是感觉意犹未尽，那么可以参考 Sprinng Security 的参考文档或者类似《Pro Spring Security》这样的相关书籍，自行深入钻研吧！

4.5.2 进一步定制 spring-boot-starter-security

除了使用 SecurityProperties 暴露的配置项（以 security.* 开头）对 spring-boot-starter-security 进行简单的配置，我们还可以通过给出一个继承了 WebSecurityConfigurerAdapter 的 JavaConfig 配置类对 spring-boot-starter-security 的行为进行更深一级的定制。

使用 WebSecurityConfigurerAdapter 的好处在于，我们依然可以使用 spring-boot-starter-security 默认约定的一些行为，只需要对必要的行为进行调整，比如：

- 使用其他的 AuthenticationManager 实例。
- 对默认 HttpSecurity 定义的资源访问的规则进行重新定义。
- 对默认提供的 WebSecurity 行为进行调整。

为了能够让这些调整生效，我们定义的 WebSecurityConfigurerAdapter 实现类一般在顺序上需要先于 spring-boot-starter-security 默认提供的配置，故此，一般配合 @Order(SecurityProperties.ACCESS_OVERRIDE_ORDER) 进行标注：

```
@Configuration
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class DemoSecurityConfiguration extends WebSecurity-
ConfigurerAdapter {

    protected DemoSecurityConfiguration() {
        super(true); // 取消默认提供的安全相关 Filters 配置
    }
```

```
    @Override
    public void configure(WebSecurity web) throws Exception {
        // ...
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // ...
    }

    // 通过Override其他方法实现对web安全的定制
}
```

WebSecurityConfigurerAdapter 其实是为我们预先设定了一个框架，并开放了有限的一些扩展点允许我们对 Web 安全相关的设定进行定制，某些场景下还是会感觉“掣肘”，或者，某些有“洁癖”的开发者，往往不想使用在某些场景下显得并非必要的默认设定。这个时候，我们可以直接实现并注册一个标注了 @EnableWebSecurity 的 JavaConfig 配置类到 IoC 容器，从而实现一种“颠覆性”的定制，即跟 spring-boot-starter-security 默认提供的 Web 安全相关配置一刀两断，完全自建：

```
@Configuration
@EnableWebSecurity
public class OverhaulSecurityConfiguration {

    @Bean
    public AuthenticationManager authenticationManager(){
        // ...
    }

    @Bean
    public AccessDecisionManager accessDecisionManager(){
        // ...
    }

    @Bean
    public SecurityFilterChain mySecurityFilterChain(){
        // ...
    }

    // 其他web安全相关组件和依赖配置
}
```

这种方案需要开发者对 Spring Security 框架本身以及 Web 安全本身有很深的理解，不到迫不得已，最好不要这么做，威力大，风险也大。

4.6 应用监控与 spring-boot-starter-actuator

所有的应用开发完成之后，其最终目的都是为了上线运行，SpringBoot 应用也不例外，而在应用运行的漫长生命周期内，为了保障其可以持续稳定的服务，我们通常需要对其进行监控，从而可以了解应用的运行状态，并根据情况决定是否需要对其运行状态进行调整，顺应需求，SpringBoot 框架提供了 spring-boot-starter-actuator 自动配置模块用于支持 SpringBoot 应用的监控。

Actuator 这个词即使翻译过来也不是很容易理解（比如翻译成“制动器；传动装置；执行机构”等），为了帮助各位更形象地理解 Actuator 是什么，笔者构思了图 4-6，希望能够有所帮助。

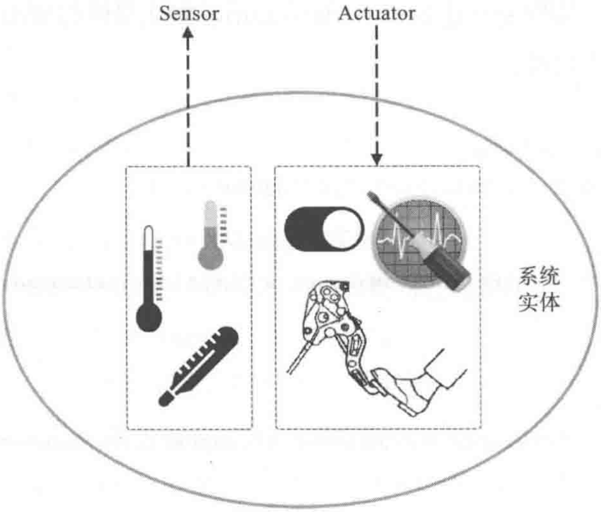


图 4-6 Sensor 和 Actuator 示意图

为了能够感知应用的运行状态，我们通常会设置一些监控指标并采集分析，这些监控指标的采集需要在应用内部设置相应的监控点，这类监控点一般只是读取状态数据，我们通常称它们为 Sensor，即中文一般称为“传感器”的东西；应用的运行状态数据通过 Sensors 采集上来之后，我们通常会有专门的

系统对这些数据进行分析和判断，一旦某个指标数据超出了预定的阈值，这往往意味着应用的运行状态在这个指标上出现了“不健康”的现象，我们希望对这个指标进行调整，而为了能够执行调整，我们需要预先在应用内部设置对应的执行调整逻辑的控制器（比如，直接关闭的开关，或者可以执行微调甚至像刹车一样直接快速拉低某个指标值的装置），这些控制器就称为 Actuator。

虽然我们日常天天在说“监控，监控”，但实际上“监”跟“控”是两个概念，Sensor 更多服务于“监”的场景，而 Actuator 则服务于“控”的场景。

spring-boot-starter-actuator 自动配置模块默认提供了很多 endpoint，虽然自动配置模块名为 spring-boot-starter-actuator，但实际上这些 endpoint 可以按照“监”和“控”划分为两类：

1. Sensor 类 endpoints

- autoconfig：这个 endpoint 会为我们提供一份 SpringBoot 的自动配置报告，告诉我们哪些自动配置模块生效了，以及哪些没有生效，原因是什么。
- beans：给出当前应用的容器中所有 bean 的信息。
- configprops：对现有容器中的 ConfigurationProperties 提供的信息进行“消毒”处理后给出汇总信息。
- info：提供当前 SpringBoot 应用的任意信息，我们可以通过 Environment 或者 application.properties 等形式提供以 info. 为前缀的任何配置项，然后 info 这个 endpoint 就会将这些配置项的值作为信息的一部分展示出来。
- health：针对当前 SpringBoot 应用的健康检查用的 endpoint。
- env：关于当前 SpringBoot 应用对应的 Environment 信息。
- metrics：当前 SprinBoot 应用的 metrics 信息。
- trace：当前 SpringBoot 应用的 trace 信息。
- mapping：如果是基于 SpringMVC 的 Web 应用，mapping 这个 endpoint 将给出 @RequestMapping 相关信息。

2. Actuator 类 endpoints

- shutdown：用于关闭当前 SpringBoot 应用的 endpoint。

- dump：用于执行线程的 dump 操作。

默认情况下，除了 shutdown 这个 endpoint（因为比较危险，如果没有安全防护，谁都可以访问它，然后关闭应用），其他 endpoints 都是默认启用的。生产环境下，如果没有启用安全防护（比如没有依赖 spring-boot-starter-security），那么，建议遵循 Deny By Default 原则，将所有的 endpoints 都关掉，然后根据具体情况单独启用某些 endpoint：

```
endpoints.enabled=false
endpoints.info.enabled=true
endpoints.health.enabled=true
...
```

所有配置项以 endpoints. 为前缀，然后根据 endpoint 名称划分具体配置项。

大部分 endpoints 都是开箱即用，但依然有些 endpoint 提供给我们进一步扩展的权利，比如健康状态检查相关的 endpoint(health endpoint)。

4.6.1 自定义应用的健康状态检查

应用的健康状态检查是很普遍的监控需求，SpringBoot 也预先通过 org.springframework.boot.actuate.autoconfigure.HealthIndicatorAutoConfiguration 为我们提供了一些常见服务的监控检查支持，比如：

- DataSourceHealthIndicator
- DiskSpaceHealthIndicator
- RedisHealthIndicator
- SolrHealthIndicator
- MongoHealthIndicator

如果这些默认提供的健康检查支持依然无法满足我们的需要，SpringBoot 还允许我们提供更多的 HealthIndicator 实现，只要将这些 HealthIndicator 实现类注册到 IoC 容器，SpringBoot 会自动发现并使用它们。

假设需要检查依赖的 dubbo 服务是否处于健康状态，我们可以实现一个 DubboHealthIndicator：

```
package com.keevol.unveilspring.chapter3.healthcheck;

import com.alibaba.dubbo.config.spring.ReferenceBean;
```

```
import com.alibaba.dubbo.rpc.service.EchoService;
import org.springframework.boot.actuate.health.AbstractHealthIndicator;
import org.springframework.boot.actuate.health.Health;

public class DubboHealthIndicator extends AbstractHealthIndicator {
    private final ReferenceBean bean;

    public DubboHealthIndicator(ReferenceBean bean) {
        this.bean = bean;
    }

    @Override
    protected void doHealthCheck(Health.Builder builder) throws Exception {
        builder.withDetail("interface", bean.getObjectType());
        final EchoService service = (EchoService) bean.getObject();
        service.$echo("hi");
        builder.up();
    }
}
```

要实现一个自定义的 HealthIndicator，一般我们不会直接实现（Implements）HealthIndicator 接口，而是继承 AbstractHealthIndicator：

```
public abstract class AbstractHealthIndicator implements HealthIndicator {

    @Override
    public final Health health() {
        Health.Builder builder = new Health.Builder();
        try {
            doHealthCheck(builder);
        }
        catch (Exception ex) {
            builder.down(ex);
        }
        return builder.build();
    }

     protected abstract void doHealthCheck(Health.Builder builder)
throws Exception;
}
```

好处就是，我们只需实现 doHealthCheck，在其中实现我们面向的具体服务的健康检查逻辑就可以了，因此，在 DubboHealthIndicator 实现类中，我们通过 dubbo 框架提供的 EchoService 直接检查相应的 dubbo 服务健康状态即可，

只要没有任何异常抛出，我们就认为检查的 dubbo 服务是状态健康的，所以，最后会通过 Health.Builder 的 up() 方法标记服务状态为正常运行。

为了完成对 dubbo 服务的健康检查，只实现一个 DubboHealthIndicator 是不够的，我们还需要将其注册到 IoC 容器中，但是一个一个单独注册太费劲了，而且还要自己提供针对某个 dubbo 服务的 ReferenceBean 依赖实例，所以，为了一劳永逸，也为了其他人能够同样方便地使用针对 dubbo 服务的健康检查支持，我们可以在 DubboHealthIndicator 的基础上实现一个 spring-boot-starter-dubbo-health-indicator 自动配置模块，即：

```
@Configuration
@ConditionalOnClass(name = {"com.alibaba.dubbo.rpc.Exporter"})
public class DubboHealthIndicatorConfiguration {
    @Autowired
    HealthAggregator healthAggregator;

    @Autowired(required = false)
    Map<String, ReferenceBean> references;

    @Bean
    public HealthIndicator dubboHealthIndicator() {

        Map<String, HealthIndicator> indicators = new HashMap<>();

        for (String key : references.keySet()) {
            final ReferenceBean bean = references.get(key);
            indicators.put(key.startsWith("&") ? key.replaceFirst("&",
"") : key, new DubboHealthIndicator(bean));
        }

            return new CompositeHealthIndicator(healthAggregator,
indicators);
    }
}
```

然后我们在 spring-boot-starter-dubbo-health-indicator 的 META-INF/spring.factories 文件中添加如下配置：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.keevol...DubboHealthIndicatorConfiguration
```

现在，发布 spring-boot-starter-dubbo-health-indicator 并依赖它就可以自动

享受到针对当前应用引用的所有 dubbo 服务进行健康检查的服务！

提示　你是否在纳闷针对 Map<String, ReferenceBean> references 的依赖注入是从哪里来的？其实 Spring 框架支持依赖注入 Key 的类型为 String 的 Map，遇到这种类型的 Map 声明（Map），Spring 框架将扫描容器中所有类型为 T 的 bean，然后以该 bean 的 name 作为 Map 的 Key，以 bean 实例作为对应的 Value，从而构建一个 Map 并注入到依赖处。

4.6.2　开放的 endpoints 才真正“有用”

不管是 spring-boot-starter-actuator 默认提供的 endpoint 实现，还是我们自己给出的 endpoint 实现，如果只是实现了放在 SpringBoot 应用的“身体内部”，那么它们不会发挥任何作用，只有将它们采集的信息暴露开放给外部监控者，或者允许外部监控者访问它们，这些 endpoints 才会真正发挥出它们的最大“功效”。

首先，spring-boot-starter-actuator 会通过 org.springframework.boot.actuate.autoconfigure.EndpointMBeanExportAutoConfiguration 将所有的 org.springframework.boot.actuate.endpoint.Endpoint 实例以 JMX MBean 的形式开放给外部监控者使用，默认情况下，这些 Endpoint 对应的 JMX MBean 会放在 org.springframework.boot 命名空间下面，不过可以通过 endpoints.jmx.domain 配置项进行更改，比如 endpoints.jmx.domain=com.keevol.management。

EndpointMBeanExportAutoConfiguration 为我们提供了一条很好的应用监控实践之路，既然它会把所有的 org.springframework.boot.actuate.endpoint.Endpoint 实例都作为 JMX Mbean 开放出去，那么，我们就可以提供一批用于某些场景下的自定义 Endpoint 实现类，比如：

```
public class HelloEndpoint extends AbstractEndpoint<String> {
    public HelloEndpoint(String id) {
        super(id, false);
    }

    @Override
    public String invoke() {
        return "Hello, SpringBoot";
```

```
    }
}
```

然后，将像 HelloEndpoint 这样的实现类注册到 SpringBoot 应用的 IoC 容器，就可以扩展 SpringBoot 的 endpoints 功能了。

注意 Endpoint 其实更适合简单的 Sensor 场景（即用于读取或者提供信息），或者简单功能的 actuator 场景（不需要行为参数），如果需要对 SpringBoot 进行更细粒度的监控，可以考虑直接使用 Spring 框架的 JMX 支持。

除了可以使用 JMX 将 spring-boot-starter-actuator 提供的（或者我们自己提供的）endpoints 开放访问，如果当前 SpringBoot 应用恰好又是一个 Web 应用，那么，这些 endpoints 还会通过 HTTP 协议开放给外部访问，与一般的 Web 请求处理一样，使用的也是 Web 应用使用的 HTTP 服务器和地址端口。因为每个 Endpoint 都有一个 id 作为唯一标识，所以，这些 endpoints 的默认访问路径其实就是它们的 id，比如 info 这个 endpoint 的 HTTP 访问路径就是 /info，而 beans 这个 endpoint 的 HTTP 访问路径则是 /beans，以此类推。

SpringBoot 允许我们通过 management. 为前缀的配置项对 endpoints 的 HTTP 开放行为进行调整：

- 使用 management.context-path= 设置自定义的 endpoints 访问上下文路径，默认直接根路径，即 /info, /beans 等形式。
- 使用 management.address= 配置单独的 HTTP 服务监听地址，比如只允许本地访问：

```
management.address=127.0.0.1
```

- 使用 management.port= 设置单独的监听端口，默认与 web 应用的对外服务端口相同，我们可以通过 management.port=8888 将管理接口的 HTTP 对外监听端口设置为 8888，但如果 management.port=-1，则意味着我们将关闭管理接口的 HTTP 对外服务。

JMX 和 HTTP 是 endpoints 对外开放访问最常用的方式，鉴于 Java 的序列化漏洞以及 JMX 的远程访问防火墙问题，建议用 HTTP 并配合安全防护将 SpringBoot 应用的 endpoints 开放给外部监控者使用。

4.6.3　用还是不用，这是个问题

endpoints 属于 spring-boot-starter-actuator 提供的主要功能之一，除此之外，spring-boot-starter-actuator 还提供了更多针对应用监控的支持和实现方案。

1. CrshAutoConfiguration 与 spring-boot-starter-remote-shell

spring-boot-starter-actuator 提供了基于 CRaSH(http://www.crashub.org/) 的远程 Shell(Remote Shell) 支持，从笔者角度来看，这是一把双刃剑，不建议在生产环境使用，因为提供给自己便利的同时，也为黑客朋友们提供了便利。如果实在要用，请加强安全认证和防护。

不过，这里我们还是会为大家分析一下 spring-boot-starter-actuator 是如何提供针对 CRaSH 的支持的。

spring-boot-starter-actuator 提供了 org.springframework.boot.actuate.autoconfigure.CrshAutoConfiguration 自动配置类，该类会在 org.crsh.plugin.PluginLifeCycle 出现在 classpath 中的时候生效，所以，只要将 CRaSH 作为依赖加入应用的 classpath 依赖就可以了，最简单直接的做法是让需要启用 CRaSH 的 SpringBoot 应用依赖 spring-boot-starter-remote-shell 自动配置模块，spring-boot-starter-remote-shell 的主要功效就是提供了针对 CRaSH 的各项依赖。

2. SpringBoot 的 Metrics 与 Dropwizard 的 Metrics

SpringBoot 提供了一套自己的针对系统指标的度量框架，这个框架的核心设计如图 4-7 所示。

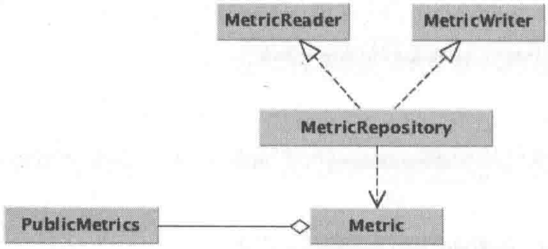


图 4-7　SpringBoot 框架的 Metrics 核心类设计示意图

基本上，我们只需关注 org.springframework.boot.actuate.endpoint.PublicMetrics 即可，它可以理解为提供一组 Metric 的集合，我们既可以通过 PublicMetrics 来汇总和管理 Metric，也可以通过 MetricRepository 来存储和管理 Metric。一旦使

用了 spring-boot-starter-actuator，只要当前 SpringBoot 应用的 ApplicationContext 中存在任何 PublicMetrics 实例，EndpointAutoConfiguration 就会将这些 PublicMetrics 采集汇总到一起，然后通过 MetricsEndpoint 将它们开放出去。

spring-boot-starter-actuator 提供的 org.springframework.boot.actuate.autoconfigure.PublicMetricsAutoConfiguration 默认会把一个 SystemPublicMetrics 开放出来，用于提供系统各项指标的度量和状态采集，另外一个就是会把当前 SpringBoot 应用的 ApplicationContext 的 org.springframework.boot.actuate.metrics.repository.MetricRepository 实例中的所有 Metric 汇总并开放出去。默认如果用户没有给出任何自定义的 MetricRepository，spring-boot-starter-actuator 会提供一个 InMemoryMetricRepository 实现，如果我们将 Dropwizard 的 Metrics 类库作为依赖加入 classpath，那么，Dropwizard Metrics 的 MetricRegistry 中所有的度量指标项也会通过 PublicMetrics 的形式开发暴露出来。

虽然 SpringBoot 提供的 metrics 框架也能帮助我们完成系统和应用指标的度量，但笔者更倾向于使用 Dropwizard 这种特定场景下比较完善的方案，从 metrics 的类型，到外围系统的集成，Dropwizard metrics 都更加成熟和完备。

3. Auditing 与 Trace

SpringBoot 的 Auditing 和 Trace 支持都遵循数据 / 事件 + Repository 的设计（如图 4-8 所示）。

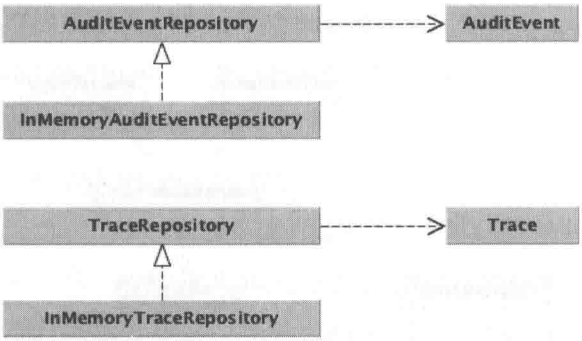


图 4-8　SpringBoot 框架 Audit 和 Trace 功能支持核心类示意图

从设计上来说是很简单清晰的，也有很好的统一性，但实际应用过程中，我们依然会更加倾向于特定场景的方案选型，比如 Auditing，我们可能只是通

过打印日志时候的 Logger 名称来区分并记录 Audit 事件，然后通过日志采集通道汇总分析就可以了，而不用非要实现一个 LogFileBasedAuditEventRepository 或者 ElasticSearchBasedAuditEventRepository 之类的实现，否则看起来难免有些“学究”气。对于 Trace 来说也是同样道理，我们可能直接使用完备的 APM 方案而不是单一或者少量 Trace 事件的记录。

4.7　本章小结

虽然我们并没有对所有的 spring-boot-starter 进行逐一剖析，但针对几个日常的 spring-boot-starter 进行剖析之后，了解和使用其他的 spring-boot-starter 也就不再神秘了。其实，一个典型的 spring-boot-starter 无外乎两点：

- 提供一个 @Configuration 配置类并通过 SpringFactoriesLoader 的配置文件 META-INF/spring.factories 注册为 org.springframework.boot.autoconfigure.EnableAutoConfiguration 标识 Key 的一个值（Value），这样该配置就可以加载到 SpringBoot 应用最终的 ApplicationContext。
- 为了实现 @Configuration 要提供的功能，我们需要在项目的 Maven 依赖中添加必要的外部依赖包，通过传递依赖（Transitive Dependencies）这一特性，使用我们的 spring-boot-starter 的项目也可以使用到这些第三方依赖包。

不过这两点并不需要同时都满足，有的 spring-boot-starter 实际上只是帮助我们引入了一些特定场景下的外部依赖，而有的 spring-boot-starter 则只提供了必要的配置，却没有引入过多的外部依赖，只是大部分 spring-boot-starter 会两者兼备。

另外，SpringBoot 提供所有 spring-boot-starter 实现时，基本都遵循“约定配置三板斧”策略：

- 建议优先使用相应 spring-boot-starter 默认的约定配置。
- 建议约定配置无法满足需要的前提下，再基于 spring-boot-starter 原有约定配置的基础上进行适当的扩展配置。
- 如果约定加适当扩展配置还是无法满足需求，则允许开发者推倒重来，基于 Spring 框架的一些原有特性直接实现自己想要的功能。

总的来说，以上三板斧策略还是比较合理的。

不管怎样，通过本章的介绍，希望大家可以对常用的一些spring-boot-starter有所理解并运用自如，同时能够“举一反三”地去了解和使用其他的spring-boot-starter。

第5章 SpringBoot微服务实践探索

Chapter 5

微服务的核心在于一个“微”字，这意味着服务的粒度更加细化，传统的“大一统”服务（Monolith）在微服务时代被打散成了一个个更加独立自主的微小服务，所以，我们如果将SOA看作一种设计策略或者指导思想和原则的话，那么，微服务就是在SOA的思想和原则指导下的一种实践方式，这种实践方式可以概括为以下几个特点：

- 在开发阶段，通过相应的微框架提供标准化的统一的开发体验和支持。
- 在发布阶段，通过标准化的形式统一发布和管理。
- 在运维阶段，通过标准化的方式统一维护数量庞大的微服务。

总的来说，微服务实践的核心竞争力就在于，我们是否围绕微服务的整个交付链路打造了一整套的支撑性工具和平台生态体系。

本章，我将跟大家一起探索如何以SpringBoot微框架为起点，围绕它打造一整套支撑其整个交付链路的工具和平台生态体系。

5.1 使用SpringBoot构建微服务

提到服务化和框架，国内大部分的研发者第一印象或许会直接联想到HSF或者Dubbo这样的服务框架，所以，我们先从中小企业服务化过程中喜闻乐见

的 Dubbo 框架开发微服务讲起吧！

5.1.1 创建基于 Dubbo 框架的 SpringBoot 微服务

Dubbo 是原阿里巴巴 B2B 平台技术部倾力打造的一款开源且优秀的面向 Java 平台的服务框架，历经三次大的迭代，从最早盲从 OSGi 的坑里跳出来，再到序列化协议和通信框架的定型，最终才完善和成型，可以说，国内开源服务化框架中无出其右者。

但是，Dubbo 研发的年代处于 Spring 框架的 XSD 时代，所以，使用上还是会更多以 XML 形式定义服务和访问服务，但好在 SpringBoot 框架在 IoC 容器的配置方式上“不挑食”，所以，我们还是可以让 Dubbo 和 SpringBoot 框架友好相处。

曾经有人问我：“我可以直接使用 Dubbo 开发微服务了呀，为什么还要用 SpringBoot？”，这是一个很好的问题。使用 SpringBoot 开发基于 Dubbo 框架的微服务的原因就是，我们将 Dubbo 微服务以 SpringBoot 形式标准化了。如此一来，我们既可以享受 SpringBoot 框架和周边的一系列研发支持，还可以用统一的形式发布、部署和运维，微服务的一个特点就是数量多，你是愿意一个人应对 1000 个相同操作接口的微服务，还是愿意一个人应对 1000 个不同操作接口的微服务？如果你跟我一样，选择的是前者，那么我们继续。

2015 年到 2016 年，美元升值预期持续升温，大家想必对汇率也比较关注，所以，我们不妨实现一个基于央行汇率的查询服务。

1. 定义汇率查询服务接口

我们新建 Maven 工程 currency-rates-api：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot</groupId>
    <artifactId>currency-rates-api</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
```

```
<name>currency-rates-api</name>
<url>http://maven.apache.org</url>


<properties>
    <java_source_version>1.8</java_source_version>
    <java_target_version>1.8</java_target_version>
    <file_encoding>UTF-8</file_encoding>
</properties>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>${java_source_version}</source>
                <target>${java_target_version}</target>
                <encoding>${file_encoding}</encoding>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-source-plugin</artifactId>
            <executions>
                <execution>
                    <id>attach-sources</id>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-plugin</artifactId>
            <configuration>
                <charset>${file_encoding}</charset>
                <encoding>${file_encoding}</encoding>
               <additionalparam>-Xdoclint:none</additionalparam>
            </configuration>
            <executions>
                <execution>
                    <id>attach-javadocs</id>
                    <goals>
```

```
                    <goal>jar</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
  </plugins>
 </build>
</project>
```

然后定义汇率查询服务接口：

```
public interface CurrencyRateService {
    ExchangeRate quote(CurrencyPair currencyPair) throws IOException;
}
```

之后通过 mvn install 或者 mvn deploy 将 currency-rates-api 发布到本地或者远程 maven 仓库。

2. 实现汇率查询服务

新建 Maven 工程 currency-rates-service：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot</groupId>
    <artifactId>currency-rates-service</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>currency-rates-service</name>
    <url>http://maven.apache.org</url>

    <properties>
        <java_source_version>1.8</java_source_version>
        <java_target_version>1.8</java_target_version>
        <file_encoding>UTF-8</file_encoding>
        <spring_version>4.1.6.RELEASE</spring_version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.keevol.springboot</groupId>
```

```
            <artifactId>currency-rates-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>dubbo</artifactId>
            <version>2.5.3</version>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring_version}</version>
            <exclusions>
                <exclusion>
                    <groupId>commons-logging</groupId>
                    <artifactId>commons-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>${spring_version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring_version}</version>
        </dependency>
    </dependencies>
</project>
```

主要关注针对 currency-rates-api 以及 Dubbo 框架的依赖定义。

项目创建完成后，我们着手实现服务接口：

```
public class CurrencyRateServiceImpl implements CurrencyRateService {

    private CurrencyRateRepository rateRepository;
```

```
    public ExchangeRate quote(CurrencyPair currencyPair) throws IOException {
        return rateRepository.get(currencyPair);
    }

    public CurrencyRateRepository getRateRepository() {
        return rateRepository;
    }

    public void setRateRepository(CurrencyRateRepository rateRepository) {
        this.rateRepository = rateRepository;
    }
}
```

其中，CurrencyRateRepository 可以根据情况给出相应的实现，比如通过直接对接银行的系统获取汇率或者通过爬取官网数据获得数据都可以，这里不做过多解释。

服务实现之后，我们需要通过 Dubbo 框架暴露出去给外部使用，所以，我们通过 Dubbo 的服务描述文件（即标准的 Spring 框架 XML 形式的配置文件）完成最终服务的对外开放：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util http://www.
springframework.org/schema/util/spring-util.xsd
        http://code.alibabatech.com/schema/dubbo
        http://code.alibabatech.com/schema/dubbo/dubbo.xsd">


    <dubbo:application name="${dubbo.application.name}" owner=
"${dubbo.application.owner}"/>

    <dubbo:protocol name="${dubbo.application.protocol.name}" port=
"${dubbo.application.protocol.port}"/>

    <dubbo:service interface="com.keevol.springboot.services.currency.
```

```
rates.CurrencyRateService" ref="serviceImpl" registry="N/A"/>

     <bean id="serviceImpl" class="com.keevol.springboot.services.
currency.rates.CurrencyRateServiceImpl"/>

</beans>
```

注意 因为只是原型示例，所以，我们没有定义更加严谨的服务注册方式（registry="N/A"），生产环境下，大家还是需要选择合适的注册服务，比如 Zookeeper。

使用 Dubbo 框架的服务不依赖传统 J2EE 的容器对外提供服务，而是以独立进程的形式对外服务，所以，我们还需要提供一个 Bootstrap 类用于启动我们的 Dubbo 服务：

```
public class Bootstrap {
    public static void main(String[] args) throws IOException {
        ApplicationContext context = new ClassPathXmlApplicationCo
ntext("spring/services.xml");
        ((AbstractApplicationContext) context).registerShutdownHook();

        System.in.read();
    }
}
```

至此，一个独立部署运行的 Dubbo 服务宣告完成。

3. 没有 SpringBoot 什么事儿

不管是否使用 SpringBoot，基于 Dubbo 框架的服务从其服务的定义，到服务的实现，都是常规而无法省略的工作，但是：

1）服务完成后，启动 main 函数里的逻辑貌似每次都要编写一样的？

2）服务完成后，以什么样的形式发布并部署？ zip 包，还是 jar 包，亦或 war 包，甚至其他形式？

考虑到这些，就会涉及 SpringBoot，一旦将 Dubbo 服务以 SpringBoot 的形式封装，Dubbo 服务就可以既享受 SpringBoot 的开发便捷性，又能以统一的形式发布和部署（比如可执行的 jar 形式）。

虽然我们无法省略和简化服务的定义和实现这些步骤，但 Dubbo 服务的

main 函数逻辑实际上是可以固化下来并且复用的，否则，每个人给出的 Dubbo 服务实现的启动类都可能不一样，进而导致运维操作不一样。

在上面我们给出的 main 函数实现中，为了阻止 Dubbo 服务的进程退出（主线程执行完毕，无其他非 daemon 线程存活），我们使用了 IO 操作来达成这一目的（System.in.read()），但实际上，这不是一个很好的做法。更好的做法是，我们设置一个服务是否关闭的开关，只有当外部调用相应管理接口将服务关闭之后，再关闭当前的 Dubbo 服务，所以，基于此思路，我们可以实现一个 ShutdownLatch：

```
public interface ShutdownLatchMBean {
    String shutdown();
}

public class ShutdownLatch implements ShutdownLatchMBean {
    protected AtomicBoolean running = new AtomicBoolean(false);

    public long checkIntervalInSeconds = 10;

    private String domain = "com.wacai.lifecycles";

    public ShutdownLatch() {
    }

    public ShutdownLatch(String domain) {
        this.domain = domain;
    }

    public void await() throws Exception {
        if (running.compareAndSet(false, true)) {
            MBeanServer mBeanServer = ManagementFactory.get-
PlatformMBeanServer();
             mBeanServer.registerMBean(this, new ObjectName(domain,
"name", "ShutdownLatch"));
            while (running.get()) {
                TimeUnit.SECONDS.sleep(checkIntervalInSeconds);
            }
        }
    }

    @Override
    public String shutdown() {
```

```
        if (running.compareAndSet(true, false)) {
            return "shutdown signal sent, shutting down..";
        } else {
              return "shutdown signal had been sent, no need again
and again and again...";
        }
    }

    public static void main(String[] args) throws Exception {
         ShutdownLatch latch = new ShutdownLatch("your_domain_for_
mbeans");
        latch.await();
    }
}
```

ShutdownLatch 默认以 JMX 的 MBean 暴露为管理接口，所以，Dubbo 服务的 main 函数可以规范为：

```
ApplicationContext context = new ClassPathXmlApplicationContext("s
pring/services.xml");
((AbstractApplicationContext) context).registerShutdownHook();

ShutdownLatch latch = new ShutdownLatch("your_domain_for_mbeans");
latch.await();
```

但这样依然需要开发人员通过“坚强的意志和超凡的感知能力“或者拷贝代码来完成这一规范行为，显然有点儿违背人性（我们都懒啊），所以，为了简化基于 SpringBoot 的 Dubbo 服务开发，我们可以将针对 Dubbo 框架的依赖以及对服务启动类的规范封装为一个 spring-boot-starter-dubbo 这样的自动配置模块，此后，要开发一个基于 SpringBoot 的 Dubbo 服务，只要依赖这一自动配置模块即可。

所以，我们新建 Maven 项目 spring-boot-starter-dubbo：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot</groupId>
    <artifactId>spring-boot-starter-dubbo</artifactId>
```

```
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-starter-dubbo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.1.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <java_source_version>1.8</java_source_version>
        <java_target_version>1.8</java_target_version>
        <file_encoding>UTF-8</file_encoding>
        <spring_version>4.1.6.RELEASE</spring_version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>

        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>dubbo</artifactId>
            <version>2.5.3</version>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
```

```
            </plugin>
        </plugins>
    </build>

</project>
```

> 提示　项目的 pom.xml 中，我们可以重点关注针对 spring-boot-starter 和 Dubbo 框架的依赖。

所有的 SpringBoot 应用启动都是基于标准的 SpringApplication.run 完成的，为了在启动类执行完成后可以阻止 Dubbo 服务进程的推出，我们需要在 SpringBoot 启动类的 main 函数最后调用 ShutdownLatch.await()，但是如果要求每个开发者在自己的 SpringBoot 启动类中调用这段代码，显然这并没有给开发者的工作带来任何简化，所以，我们选择使用 CommandLineRunner 来完成针对 ShutdownLatch.await() 的调用工作，前面我们已经介绍过 CommandLineRunner，任何注册到 SpringBoot 应用的 CommandLineRunner 都将在 SpringApplication.run 执行完成后再执行，恰好符合我们当前场景需要，所以，就有了：

```
public class DubboServiceLatchCommandLineRunner implements
CommandLineRunner {

    private String domain = "com.keevol.services.management";

    @Override
    public void run(String... args) throws Exception {
        ShutdownLatch latch = new ShutdownLatch(getDomain());
        latch.await();
    }


    public String getDomain() {
        return domain;
    }

    public void setDomain(String domain) {
        this.domain = domain;
    }
}
```

然后我们需要将 DubboServiceLatchCommandLineRunner 注册到 SpringBoot

应用的容器之中，所以，定义一个 JavaConfig 类如下：

```
@Configuration
@Order
public class DubboAutoConfiguration {

    protected Logger logger = LoggerFactory.getLogger(DubboAutoConf-
iguration.class);

      @Value("${shutdown.latch.domain.name: com.keevol.services.
management}")
    private String shutdownLatchDomainName;

    @Bean
    @ConditionalOnClass(name = "com.alibaba.dubbo.rpc.Exporter")
    public DubboServiceLatchCommandLineRunner configureDubboService-
LatchCommandLineRunner() {
        logger.debug("DubboAutoConfiguration enabled by adding Dubbo-
ServiceLatchCommandLineRunner.");
        DubboServiceLatchCommandLineRunner runner = new DubboServi-
ceLatchCommandLineRunner();
        runner.setDomain(shutdownLatchDomainName);
        return runner;
    }
}
```

我们使用 @Order 标注了该配置类以保证 DubboServiceLatchCommandLine Runner 在最后执行，以避免阻塞其他逻辑的执行。

万里长征走到了最后一步，要实现一个 SpringBoot 的自动配置模块，我们需要将 DubboAutoConfiguration 配置类通过 META-INF/spring.factories 配置文件注册并发布：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.
keevol.springboot.dubbo.autoconfigure.DubboAutoConfiguration
```

至此，我们只要通过 mvn install 或者 mvn deploy 将 spring-boot-starter-dubbo 发布到本地或者远程 Maven 仓库，以后开发 Dubbo 服务就只需要在服务的项目依赖中添加如下依赖：

```
<dependency>
    <groupId>com.keevol.springboot</groupId>
    <artifactId>spring-boot-starter-dubbo</artifactId>
    <version>1.0.0</version>
```

```
</dependency>
```

然后以标准的 SpringApplication 加载 Dubbo 服务的配置并启动就可以了：

```
@SpringBootApplication
public class DubboWithSpringbootApplication {
    public static void main(String[] args) {
        SpringApplication application = new SpringApplication(Dubb-
oWithSpringbootApplication.class, "classpath*:/spring/**/*.xml");
        application.run(args);
    }
}
```

当然，规范 Dubbo 依赖以及服务的启动逻辑只是使用 SpringBoot 获得的好处之一，最主要的，我们提供了一种基于 SpringBoot 的标准化的 Dubbo 服务开发和发布实践，这对于海量微服务的开发和运维来说是很重要的。

5.1.2 使用 SpringBoot 快速构建 Web API

Dubbo 框架现在在国内的中小企业当中已经成为 Java 生态下服务化的事实标准，出现这种状态的原因很多，比如 Dubbo 框架设计优秀、文档和资料丰富、配置灵活、特性丰富等，但最主要的，我认为是 Java 开发人员对速度这一因素的痴迷。

不可否认，Dubbo 框架设计和实现之初就将其自身定位为一款基于 TCP 长连接通信的高性能服务治理框架，但是，对于很多中小企业来说，不管从速度还是并发度，根本就没有到非要使用像 Dubbo 这样基于 TCP 长连接服务框架的程度。笔者认为，不分场景和现状盲目选型 Dubbo 框架，或许就是 Dubbo 框架成为 java 生态下服务治理框架事实标准的原因。

Dubbo 框架虽然有很多优点，也确实面向高强度的互联网应用场景，且在多家知名的互联网企业的生产环境得到验证，但也并非没有缺点：

- 只限于 Java 应用之间的服务调用。
- 服务访问方需要依赖 API 以及关联依赖，在很多场景下导致依赖管理混乱的问题。
- 核心项目人员转岗或者离职之后，项目不再有人专职投入维护和升级，虽然功能够用，但任何一个开源项目无推动者，无活跃社区的情况下，其生命走向只有一个，而这几乎是国内开源项目的共同宿命。

作为一名理性的研发人员，在项目技术选型的时候，需要综合考虑多种方案的优缺点，并根据现状进行权衡，实际上，对于大部分项目来说，性能可能并非技术选型的核心因素，开放和互通或许才是。

是要以互通性作为核心因素进行技术选型并构建一套开放繁荣的生态体系，还是以性能为核心因素进行技术选型构建一套封闭高效的生态体系，需要大家灵活把握，而本节我们将更多以 Web API 的形式，向大家展示如何基于 SpringBoot 构建一套开放、互通、稳定的 Web API 微服务体系。

使用 SpringBoot 构建 Web API 有几种选择，要么使用 spring-boot-starter-jersey 构建 RESTful 风格的 Web API，要么选择 spring-boot-starter-hateoas 构建更加有关联性和相对“智能”的 Web API，但笔者认为这些都有点儿“阳春白雪”，对于大部分开发人员来说，HTTP 协议的 GET 和 POST 是直觉上最自然的选择，所以，我们选择使用最“下里巴人”的方式来构建 Web API ⊖。

Web API 强调统一和互通，所以，首先我们需要定义一套内外认知一致的 Web API 开发和访问规范，在 JSON 盛行、社群庞大的背景下，我们的 Web API 方案采用 JSON 作为数据交互格式并定义统一的协议格式，然后通过 HTTP 以及周边支持完成微服务的对外服务和开放访问。

1. 定义 Web API 规范

首先从服务访问的交互上来说，我们可以选择较为纯粹的 JSON RPC Over HTTP 的方式，如图 5-1 所示。

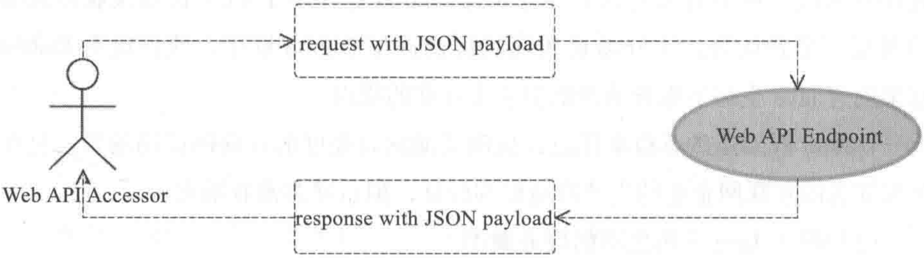


图 5-1 JSON RPC Over HTTP 示意图

⊖ 当十年前 Hibernate 作为 ORM 的事实标准在 Java 的数据访问领域“叱咤风云”的时候，在是否使用 Hibernate 的各种高级映射特性问题上，存在两派，一派也是“阳春白雪”派，一派则是“下里巴人”派，前者极力推崇应该使用那些高级映射特性，而后者则只使用了 Hibernate 最基本的功能，谁对谁错并不重要，只是，如果大家对我们的 Web API 选型有些不理解，可以适当读下“历史”，并对比讨论一下，或许会豁然开朗。

也可以选择约束相对松一些的 RPC Over HTTP 方式，如图 5-2 所示。

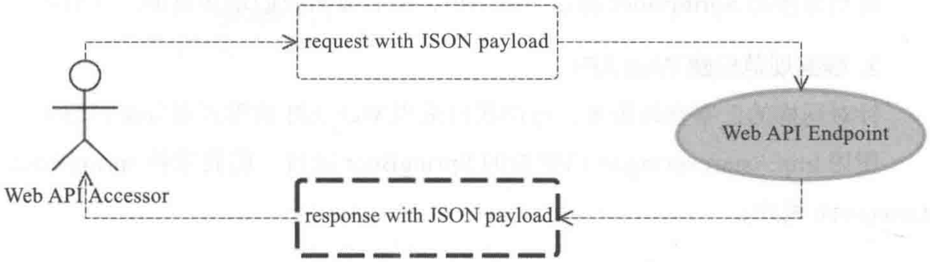


图 5-2　一般意义上基于 HTTP 的 RPC 交互示意图

相对于纯粹的 JSON RPC Over HTTP 方案，后者对请求格式不做任何限制（所以也同样支持纯粹 JSON 形式的请求格式），只对响应（Response）做 JSON 格式上的统一规定，好处是，客户端各种工具都能够很好的支持，服务器端 SpringMVC 也可以少做 HttpMessage 转换，给服务的开发者和访问者都提供了比较灵活的操作余地，至于请求的类型差异，我们可以通过配套生成的 API 文档进行补足。

不管怎么样，我们选择基于后一种方案进行说明，现在剩下的主要工作就是定义服务响应格式，只有规范和统一了服务的响应格式，才能让内部和外部的服务访问者形成统一的认知，以同样的方式“复制”对我们提供的任何 Web API 的访问行为，减少用户的接入成本，所以，姑且我们简单规定一个服务的响应格式如下：

```
{
    "code" : 1,
    "error" : "XXXXX",
    "data" : {
        ...
    }
}
```

其中，code 表示调用结果的状态，0 表示成功，非 0 表示失败，并且失败情况下 error 字段将提供对应的错误信息描述，data 字段用于规范定义特定于 Web API 的响应内容。

有了这样的规范定义，不同的开发者就可以根据情况选择打造对应的工具或者 SDK 了。而 Web API 的服务提供者也同样可以根据该规范考虑如何简化

Web API 的开发，或者通过约束减少规范认知不足可能导致的问题。

既然是使用 SpringBoot 构建 Web API，那么显然我们现在更加关注后者。

2. 根据规范构建 Web API

针对同样的汇率查询服务，这回我们采用 Web API 的形式对外提供服务。

使用 http://start.spring.io 构建新的 SpringBoot 项目，使其依赖 spring-boot-starter-web 模块：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot.chapter4</groupId>
    <artifactId>currency-webapi</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>currency-webapi</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.1.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>com.keevol.springboot</groupId>
```

```
            <artifactId>currency-rates-service</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

因为我们已经实现了 CurrencyRateService，所以，可以直接将其作为项目依赖的一部分（当然，这样也让我们的 Web API 看起来更像一个适配网关了）。

我们直接使用 SpringMVC 构建对应的 Controller 对外提供 Web API 的访问如下：

```
@Controller
public class CurrencyRateQueryController {
    @Autowired
    private CurrencyRateService currencyRateService;

    @RequestMapping(value = "/", method = RequestMethod.GET)
    @ResponseBody
    public ExchangeRate quote(String symbol) throws IOException {
        return currencyRateService.quote(CurrencyPair.from(symbol));
    }
}
```

spring-boot-starter-web 默认会提供一系列的 HttpMessageConverter 用于对请求参数和响应结果做类型转换，所以，ExchangeRate 类型将通过默认 HttpMessageConverter 序列中的 MappingJackson2HttpMessageConverter 转换成对应的 JSON 响应结果，类似于：

```
{
    currencyPair: {
        symbol: "USD/CNY"
    },
```

```
    bidPrice: 6.67,
    askPrice: 6.56
}
```

整个 Web API 的功能流程算是跑通了，但跟我们之前定义的 Web API 规范却没有关系，所以，下一步我们要做的事情就是在此基础上规范 HTTP 响应格式，使其遵循我们之前定义的 Web API 规范，从而任何访问我们提供的 Web API 访问者都可以相同的认知使用这些 Web API，进而也可以打造和沉淀相应的工具或者类库。

注意 我们定义的 Web API 规范并非最优，也并非必要，如果团队成员的认知差不多，那么直接使用 HTTP Status Code 结合直接的值类型响应就可以了，我们给出的 Web API 规范考虑了更多因素后做出的一个折中方案，但任何方案设计是否完美并非最主要的，执行才是。

要开发符合我们的 Web API 规范的 Web API，最少有两种方案可以选择：

- 显式的强类型封装方式（explicit type wrapper）
- 隐式的自动转换方式（implicit conversion）

显式的强类型封装方式的出发点是说，既然 spring-boot-starter-web 已经提供了 MappingJackson2HttpMessageConverter 用于对象类型到 JSON 的类型转换，那么，我们只要提供针对 Web API 规范的 Java 对象类型作为所有 Web API 处理方法的返回值就可以了，比如：

```
public class WebApiResponse<T> {

    public static final int SUCCESS_CODE = 0;
    public static final int ERROR_CODE = 1;

    private int code;
    private String error;
    private T data;

    // getters, setters, toString(), etc.
}
```

然后，所有的 Web API 的处理方法统一定义为返回 WebApiResponse 作为结果类型：

```
@RequestMapping(value = "/", method = RequestMethod.GET)
@ResponseBody
public WebApiResponse<ExchangeRate> quote(String symbol) throws
IOException {
    WebApiResponse<ExchangeRate> response = new WebApiResponse<>();
    response.setCode(WebApiResponse.SUCCESS_CODE);
    response.setData(currencyRateService.quote(CurrencyPair.from(symbol)));
    return response;
}
```

不过，这种模式过于强调规范的管控，对开发者来说不是太友好，即使我们通过 Builder 模式来简化 WebApiResponse 的构造过程，比如：

```
public class WebApiResponse<T> {

    public static final int SUCCESS_CODE = 0;
    public static final int ERROR_CODE = 1;

    private int code;
    private String error;
    private T data;


    public static <T> WebApiResponse<T> success(T data) {
        WebApiResponse<T> response = new WebApiResponse<>();
        response.setCode(SUCCESS_CODE);
        response.setData(data);
        return response;
    }

    public static <T> WebApiResponse<T> error(String errorMessage) {
        return WebApiResponse.<T>error(errorMessage, ERROR_CODE);
    }
    // ...
}
    @RequestMapping(value = "/", method = RequestMethod.GET)
    @ResponseBody
    public WebApiResponse<ExchangeRate> quote(String symbol)
throws IOException {
        return WebApiResponse.success(currencyRateService.quote
(CurrencyPair.from(symbol)));
    }
```

但从 API 的使用者角度来看，这种设计并非最优，最好的方式其实应该是隐式的自动转换方式。在隐式的自动转换方式下，用户的 Web API 处理方法定

义保持不变，直接返回最原始的值类型（比如 ExchangeRate）：

```
@RequestMapping(value = "/", method = RequestMethod.GET)
@ResponseBody
public ExchangeRate quote(String symbol) throws IOException {
    return currencyRateService.quote(CurrencyPair.from(symbol));
}
```

通过在框架层面对原始的值类型进行符合规范行为的封装，最终返回给用户的响应结果“自动”的或者说以“不打扰 API 开发者”的形式变成了符合我们 Web API 规范的响应结果形式。

要达到隐式的自动转换方式的效果，最简单粗暴的做法就是完全覆盖 Web 应用的配置，只配置一个自定义处理 JSON 转换的 HttpMessageConverter 实现，比如：

```
public class JsonHttpMessageConverter extends AbstractHttpMessage-
Converter<Object> {

    @Override
    protected boolean supports(Class<?> clazz) {
        return !clazz.isPrimitive();
    }

    @Override
    protected Object readInternal(Class<?> aClass, HttpInputMessage
httpInputMessage) throws IOException, HttpMessageNotReadableException
{
        return null;
    }

    @Override
     protected void writeInternal(Object o, HttpOutputMessage
httpOutputMessage) throws IOException, HttpMessageNotWritableException {
         httpOutputMessage.getHeaders().add("Content-Type",
"application/json");
        // 其他 header 设置

        // toJson() 方法中可以使用 jackson 或者 fastjson 等类库完成对象到
json 的转换
        httpOutputMessage.getBody().write(toJson(o));
        httpOutputMessage.getBody().flush();
    }
}
```

但是，这会导致一些问题或者不便：

1）打破了 SpringBoot 对 SpringMVC 的完备支持，对于大部分已经很熟悉 SpringMVC 框架中各种功能和类库使用的读者来说，这些可能不再有效；

2）SpringBoot 提供的 spring-boot-starter-web 模块的默认配置项都不再有效，比如 SpringBoot 参考文档中的 spring.jackson.serialization.indent_output = true 之类的配置项，这显然是在舍弃已有的良好文档和功能支持；

3）因为现在只有一个 HttpMessageConverter 处理单一类型的 Web 请求和响应，如果同一项目中有类似视图渲染的需求，则无法满足需求。

所以，为了能够不打破开发者对 SpringMVC 框架以及 SpringBoot 提供的 Web 应用各项功能支持的认知，最稳妥的做法是，在 SpringBoot 原有 Web 应用默认配置的基础上增加新的 HttpMessageConverter，专门处理 Web API 响应结果使其符合我们的 Web API 规范形式。

要达到这个目的，我们可以提供自定义的配置：

```
@Configuration
public class WebApiConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void extendMessageConverters(List<HttpMessageConvert-
er<?>> converters) {
        // 添加或者插入我们自定义的 HttpMessageConverter 实现类
        // converters.add(converter) 或者 converters.add(0, converter)
    }
}
```

extendMessageConverters 属于已经添加过默认 HttpMessageConverter 序列的参数（比如针对 String 的 HttpMessageConverter，或者针对 byte[] 的 HttpMessageConverter 等），所以，我们只要在其基础上添加或者插入我们的 HttpMessageConverter 实现类就可以了。不过，这里有一个比较尴尬的地方，这可能也是 Spring 框架多处设计中都存在的尴尬，即循环条件判断应用哪个类的时候，条件判断 API 开放不足：

```
for(HttpMessageConverter converter: converters){
    if(converter.canWrite(clazz, media)) {
        converter.write(..);
    }
}
```

在 HttpMessageConverter 的场景中就是，我们只能根据目标对象的类型以及 mediaType 来判断是否应该使用当前这个 HttpMessageConverter，如果需要在这两种判断条件都相同的情况下，还要根据其他条件来判定是否应该使用当前 HttpMessageConverter，此时这种设计显然就无法满足需求了。尴尬之处就在于此。

对于我们的 Web API 规范这个实现场景来说，如果想继续享受原有的 MappingJackson2HttpMessageConverter 提供的功能和配置，就不得不继承并覆写（Override）相应方法，而不是略过 MappingJackson2HttpMessageConverter，然后在另一个 HttpMessageConverter 中只是必要的时候引用它（组合优于继承）。

不管怎么样，我们推荐使用隐式的自动转换方式为用户提供透明的 Web API 规范行为。

3. Web API 的短板和补足

相对于 Dubbo 这种强类型的服务框架，Web API 没有强类型支持（Not Typesafe），在开发过程中，自然也无法享受到像 IDE 自动提示之类的功能，所以，对于 Web API 的使用者来说，需要与 Web API 的提供者沟通之后才能知道如何访问 Web API 的详细信息，比如应该传哪些参数，返回的响应结果又应该是什么格式的。

为了缓解这个问题，我们可以使用自动根据代码元信息生成 API 文档的方式来补足这块短板，像 Swagger(http://swagger.io/) 这样的项目，已经是比较成熟的 API 文档方案了。

不过，让每一个 Web API 项目都自己去初始化 API 文档相关的设置显然并不是很好的用户体验，为了服务到位，我们可以遵循 SpringBoot 的行事风格，新建一个 spring-boot-starter-webapi 这样的自动配置模块，其提供的主要特性包括但不限于：

- 提供针对我们 Web API 规范的功能支持，即提供显式的强类型封装方式或者隐式的自动转换方式的功能实现。
- 提供 API 文档相关功能的配置和设置。
- 提供统一的 Web API 访问错误处理逻辑。

这样，任何 Web API 的开发者和提供者只要新建 SpringBoot 应用，然后依赖 spring-boot-starter-webapi，就可以自动享有以上所有特性支持了。

以下是一个 spring-boot-starter-webapi 原型项目的 pom.xml 定义：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starters</artifactId>
        <version>1.2.5.RELEASE</version>
    </parent>
    <groupId>com.keevol.springboot</groupId>
    <artifactId>spring-boot-starter-webapi</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-starter-webapi</name>
    <url></url>

    <properties>
        <java.version>1.8</java.version>
        <file.encoding>UTF-8</file.encoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                    <encoding>${file.encoding}</encoding>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
```

```
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
            <version>2.1.2</version>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger-ui</artifactId>
            <version>2.1.2</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>${servlet-api.version}</version>
        </dependency>
    </dependencies>
</project>
```

以及对应的 JavaConfig 配置类示例：

```
@Configuration
@EnableSwagger2
@ComponentScan("com.wacai.springboot.webapi.errors")
@AutoConfigureAfter(WebMvcAutoConfiguration.class)
public class WebApiAutoConfiguration extends WebMvcConfigurerAdapter {

     protected Logger logger = LoggerFactory.getLogger(WebApiAuto-
Configuration.class);

    @Value("${springfox.api.group:[your api group name]}")
    private String apiGroupName;

    @Value("${springfox.api.title:[set a api title via 'springfox.
api.title']}")
    private String title;
     @Value("${springfox.api.description:[add your api description
via 'springfox.api.description'}]")
    private String desc;
     @Value("${springfox.api.version:[set specific api version via
'springfox.api.version'}]")
    private String version;
     @Value("${springfox.api.termsOfServiceUrl:[set termsOf-
ServiceUrl via 'springfox.api.termsOfServiceUrl']}")
    private String termsOfServiceUrl;
    @Value("${springfox.api.contact:[set contact via 'springfox.api.
```

```
contact'}]")
    private String contact;
    @Value("${springfox.api.license:Your WebAPI License}")
    private String license;
    @Value("${springfox.api.licenseUrl:http://keevol.com}")
    private String licenseUrl;

    @Autowired
    private TypeResolver typeResolver;

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .groupName(apiGroupName)
                .apiInfo(new ApiInfo(title, desc, version, termsOf-
ServiceUrl, contact, license, licenseUrl))
                .select()
                .apis(RequestHandlerSelectors.any())
                .paths(excludedPathSelector())
                .build().pathMapping("/")
                .directModelSubstitute(Date.class, String.class)
                .genericModelSubstitutes(ResponseEntity.class)
                .alternateTypeRules(newRule(typeResolver.resolve
(DeferredResult.class, typeResolver.resolve(ResponseEntity.class,
WildcardType.class)), typeResolver.resolve(WildcardType.class)))
                .useDefaultResponseMessages(false)
                .globalResponseMessage(RequestMethod.GET,
newArrayList(new ResponseMessageBuilder().code(500).message("服务出
错啦～").responseModel(new ModelRef("Error")).build()))
                .forCodeGeneration(true);
    }

    // ...

}
```

关于如何将 WebApiAutoConfiguration 配置到 META-INF/spring.factories 并发布项目则不再赘述。有了 spring-boot-starter-webapi 之后，Web API 形式的微服务开发者所要做的仅仅是把它加为项目依赖：

```
<dependency>
    <groupId>com.keevol.springboot</groupId>
    <artifactId>spring-boot-starter-webapi</artifactId>
    <version>1.0.0-SNAPSHOT</version>
```

```
</dependency>
```

然后像往常那样写 SpringMVC 的 @Controller 或者 @RestController 就可以了，现在，我们可以直接享受 API 文档的自动生成（如图 5-3 所示），是不是很棒？

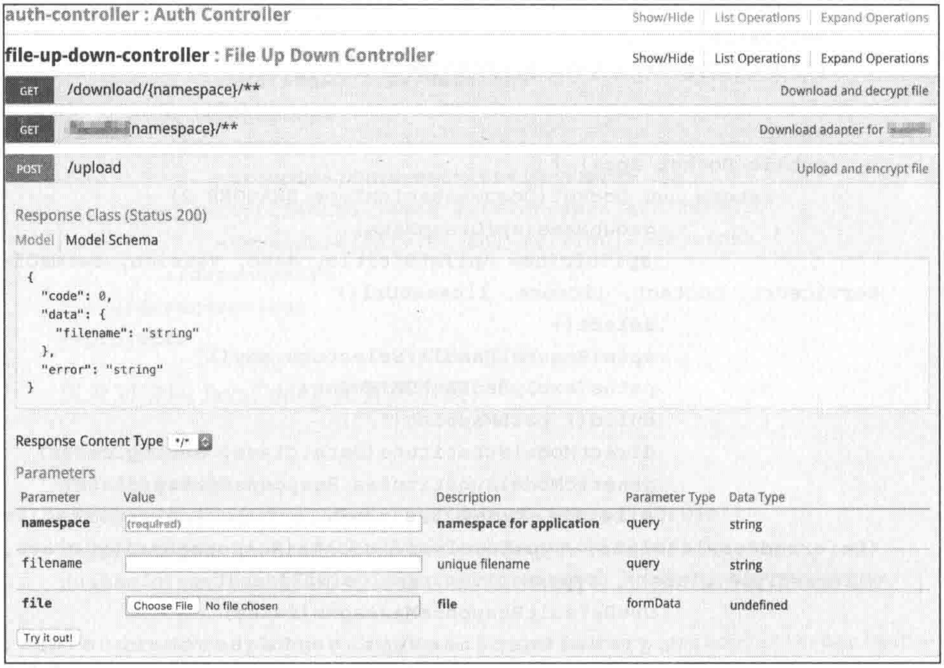


图 5-3 API 文档效果图

5.1.3 使用 SpringBoot 构建其他形式的微服务

目前为止，我们介绍了使用 SpringBoot 开发基于 Dubbo 框架的微服务，以及使用 SpringBoot 开发 Web API 形式的微服务，貌似两种都是 RPC 形式的微服务形式，但并非所有微服务都应该是 RPC 形式的，而且 SpringBoot 也并没有对微服务的具体服务形式进行严格规定，正如我们之前所说的那样，SpringBoot 只是提供了一种微服务的标准化实践方式而已。

区别于 RPC 形式对外提供服务的微服务，这次我们尝试开发另一种微服务，这种微服务只响应外部事件并做处理，比如在外汇交易系统或者股票交易系统中，我们可以设置一个个独立的微服务，在接收到市场行情事件或者数据

的时候，构建和归档蜡烛图数据，架构设计如图 5-4 所示。

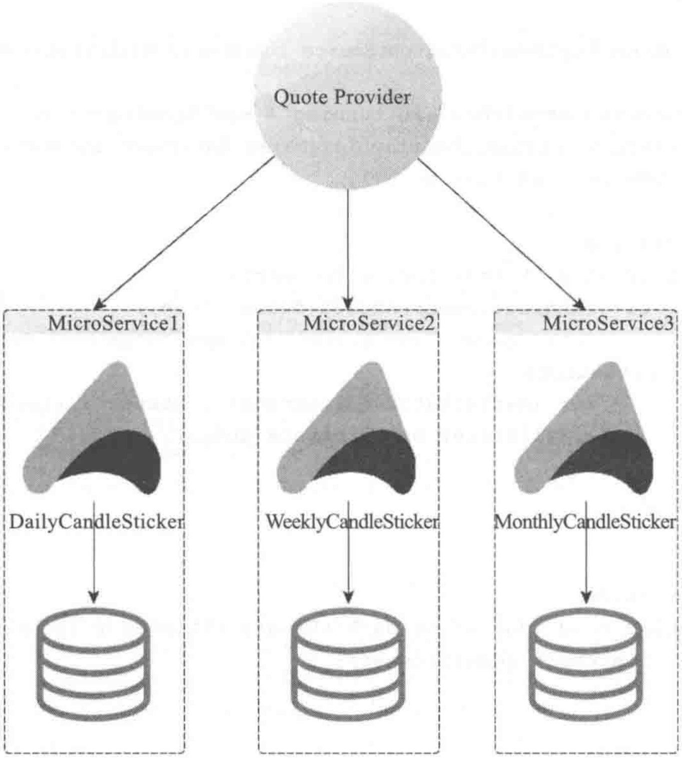


图 5-4　基于 Actor 的蜡烛图数据计算微服务示意图

不考虑泛型以及服务的通用性，我们尝试使用 akka 框架的 Actor 模型来开发一个生成日线蜡烛图的 SpringBoot 微服务。

为了简化原型实现，我们将日线蜡烛图微服务（DailyCandleSticker）抽象为两部分。

第一部分是针对市场数据来源的抽象，我们将从上游供应商处获得市场数据的行为抽象为：

```
public interface MarketDataEventSource {
    void start() throws Throwable;

    void addListner(MarketDataEventListener listner);

    void shutdown() throws Throwable;
}
```

然后根据不同的供应商以及他们使用的不同交互协议类型，提供不同的实现类，比如：

```
public class ToyMarketDataEventSource implements MarketDataEventSource {

    protected AtomicBoolean running = new AtomicBoolean(false);
    protected CopyOnWriteArrayList<MarketDataEventListener> listeners
= new CopyOnWriteArrayList<>();

    @Override
    public void start() throws Throwable {
        if (running.compareAndSet(false, true)) {
            Quote quote = new Quote(); // get quote from remote source
in real situations
            for (MarketDataEventListener listener : listeners) {
                listener.onMarketData(quote);
            }
        }
    }

    @Override
    public void addListner(MarketDataEventListener listener) {
        listeners.add(listener);
    }

    @Override
    public void shutdown() throws Throwable {
        if (running.compareAndSet(true, false)) {
            System.out.println("clean up");
        }
    }
}
```

MarketDataEventSource 在被 start() 之前，可以向其添加一系列的市场数据监听器，用于处理市场数据，一旦 MarketDataEventSource 被 start()，则通过特定技术（比如 Server Sent Event，或者特定消息中间件，比如 Kafka）获取到的市场数据就可以传递给相应的 MarketDataEventListener 进行处理：

```
public interface MarketDataEventListener {
    void onMarketData(Quote quote) throws Throwable;
}
```

该微服务原型代码的第二部分才是我们要演示的核心，即针对获取到的市

场数据构建日线蜡烛图。我们使用 Akka 的 Actor 来构建处理单元，一个简单的定义如下：

```
class CandleSticker(symbol: String, context: ApplicationContext)
extends Actor with ActorLogging {

  val candleStickRepository = context.getBean(classOf[CandleStickR
epository])

  var openPrice: Option[BigDecimal] = None
  var closePrice: Option[BigDecimal] = None
  var highestPrice: Option[BigDecimal] = None
  var lowestPrice: Option[BigDecimal] = None


  override def receive = {
    case MarketOpen => {
      // do sth. on market open event if necessary
    }
    case MarketClose => {
      val candleStick = new CandleStick
      candleStick.openPrice = openPrice.get
      candleStick.closePrice = closePrice.get
      candleStick.highestPrice = highestPrice.get
      candleStick.lowestPrice = lowestPrice.get
      candleStickRepository.store(candleStick)
      resetCandleSticker()
    }
    case quote: Quote => {
      if (openPrice.isEmpty) {
        openPrice = Some(quote.price)
        highestPrice = openPrice
        lowestPrice = openPrice
      } else if (quote.price > highestPrice.get) {
        highestPrice = Some(quote.price)
      } else {
        // more compare and set
      }
    }
    case _ => // handle unexpected conditions
  }

  def resetCandleSticker(): Unit = ???
}
```

可以看到 CandleSticker 会根据接收到的不同类型的市场数据（比如 MarketOpen 和 Quote 等）做出相应的处理，并在市场关闭事件（MarketClose）到达之后，构建并存储当日的日线蜡烛图数据。

注意 如果我们想同时使用 Actor 以及 Spring IoC，那么，我建议所有 Actor 的定义都像以上 CandleSticker 定义那样，传递 ApplicationContext 给 Actor 定义，让其使用 DL(Dependency Lookup) 的模式来使用容器中提供的各种服务依赖（类似上例代码中的 val candleStickRepository = context.getBean(classOf[CandleStickRepository]))，主要原因在于 Actor 的生命周期是短暂而易变的，而 ApplicationContext 以及其中的各项依赖服务则不然，如果硬要使用依赖注入的方式（DI - Dependency Injection)，难免有点儿“削足适履”之嫌。

当以上两部分抽象和实现都完成之后，剩下要做的，就是将这两部分通过 SpringBoot 封装到一起，下面就是我们的 SpringBoot 应用启动类：

```
@Configuration
public class CandleStickerBootstrap {

    @Bean
    public CandleStickRepository candleStickRepository() {
        return ...;
    }
    // other dependency configurations if any

    public static void main(String[] args) throws Throwable {
        ApplicationContext context = SpringApplication.run
(CandleStickerBootstrap.class, args);

        final ActorSystem actorSystem = ActorSystem.apply();

        List<Object> actorArgs = new ArrayList<>();
        actorArgs.add("USD/CNY");
        actorArgs.add(context);
        final ActorRef candleSticker = actorSystem.actorOf (Props.
apply(CandleSticker.class, JavaConversions.asScalaBuffer(actorArgs)));

        final MarketDataEventSource eventSource = new Market-
DataEventSourceImpl();
```

```
        eventSource.addListner((Quote quote) -> candleSticker.
tell(quote, ActorRef.noSender()));
        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                try {
                    eventSource.shutdown();
                } catch (Throwable throwable) {
                    // log warnings
                }
                candleSticker.tell(PoisonPill.getInstance(), ActorRef.
noSender());
                actorSystem.terminate();
            }
        });

        eventSource.start();
    }
}
```

虽然用 Java 代码来写 Akka Actor 看起来有些繁琐（使用 Scala 编写和使用 Akka Actor 要简洁的多），但不会掩盖一个标准的 SpringBoot 应用启动类的常见逻辑，我们依然是通过 SpringApplication.run 来启动当前的 SpringBoot 应用，只不过，在日线蜡烛图微服务的场景下，SpringApplication.run 给予的唯一功能，其实只是帮忙构建了一个包含了必要服务依赖的 ApplicationContext，拿到 ApplicationContext 实例引用之后，我们要继续构建相应的 CandleSticker Actor，然后将其作为一个 MarketDataEventListener 的处理逻辑单元与 MarketDataEventSource 挂接，然后启动 MarketDataEventSource，从而完成整个日线蜡烛图微服务的启动工作。

注意 假如我们发现使用 Akka Actor 和 SpringBoot 的应用在 SpringApplication.run 之后存在一些共性的逻辑，也可以提供一个 CommandLineRunner 对共性逻辑进行封装，然后构建一个 spring-boot-starter-akka-actor 自动配置模块，从而造福一方。

至此，我们介绍了三种基于不同框架和技术的 SpringBoot 微服务实现，希望大家可以通过对比学习，对如何使用 SpringBoot 构建微服务有更深入的理解并付诸实践。

5.2 SpringBoot 微服务的发布与部署

基于 SpringBoot 的微服务开发完成之后，现在到了把它们发布并部署到相应的环境去运行的时候了。

SpringBoot 框架只提供了一套基于可执行 jar 包（executable jar）格式的标准发布形式，但并没有对部署做过多的界定，而且为了简化可执行 jar 包的生成，SpringBoot 提供了相应的 Maven 项目插件：

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <!-- 其他插件定义 -->
    </plugins>
</build>
```

然后只要我们运行 mvn package，当前 SpringBoot 项目就会被打包成一个包含了其所有项目依赖以及该项目本身的可执行 jar 包，通过 scp 或者 rsync 等方式将这个可执行 jar 包部署到目标环境的服务器之后，就可以通过 java -jar your-project.jar 启动 SpringBoot 应用了。⊖

整个流程看起来很简单，也很符合大部分开发人员的认知，但是，相对于一套较为严谨的软件交付流程来说，以上流程则难免过于粗糙了。

软件的发布和部署可以有多种不同的形式，这更多由软件项目的属性决定：

- 这个项目使用的是什么语言？
- 这个项目属于类库项目还是可独立运行的项目？
- 这个项目是面向什么平台和环境的项目？

此外，我们希望使用什么样的形式进行软件的交付，这里则涉及生态管理以及技术选型的喜好等因素，所以，为了降低讲解的复杂度，我们还是先将发

⊖ 如果纯粹只是为了好玩，或者 Java 的死忠，既然 SpringBoot 微服务默认是打包为可执行 jar 包直接执行，那么，如果愿意，我们可以 SpringBoot 微服务的形式写一些命令行工具并发布为可执行 jar 包运行，只是，这种做法相对于动态脚本语言来说，实在有些太“重”了。

布和部署分开来说吧。

首先，大家应该都知道，发布并不等于部署，这是两个阶段的事情，如图 5-5 所示。

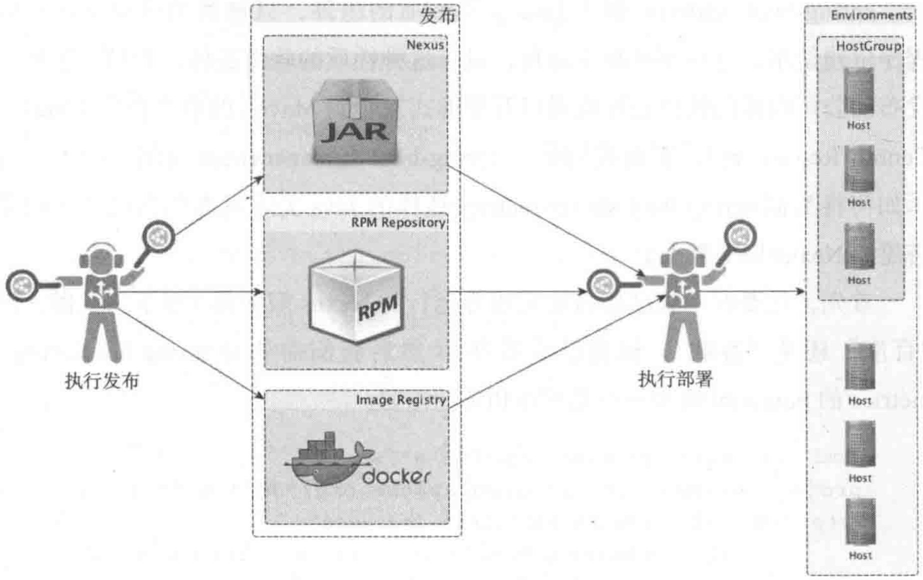


图 5-5 发布与部署示意图

发布一般是将项目以指定的格式打包成某种可直接交付的形式，然后放置到预先指定的交付地点，比如对于 Java 类库（Java Library）来说，我们一般将其打包成 jar 包，然后 mvn deploy 到公司内部的 Maven 仓库中（Maven Repository），像 Nexus Repository Manager 或者 JFrog Artifactory 以及 Apache Archiva；而对于可独立运行的程序，比如 SpringBoot 微服务或者一般的 Java Standalone 程序，我们既可以将它们打包成 RPM、DEB 等面向特定目标系统的发布形式，也可以将它们制作成一个个的 docker images，然后将制作完成的发布成品存储到相应的仓库中（Repository）去。

部署一般紧接着发布完成之后进行，它的主要职能就是将已经发布好的成品从仓库中拿出来，然后分发到目标环境的指定资源池（比如物理机结点，虚拟机结点，docker 宿主机等），并最终启动服务。软件成品分发的手段和工具可以有很多种，从最常见的 scp、rsync，到 Chef、Puppet，进而再到最新的 saltstack、ansible 等，一般根据团队对这些工具的把控力度和喜好进行选型。

下面我们就几种典型的发布和部署形式跟大家一起探索相应的实践。

5.2.1 spring-boot-starter 的发布与部署方式

spring-boot-starter(s) 属于 Java 类库性质的组件，只被其他可独立运行的程序依赖使用，自身不可独立运行，对于这种性质的软件实体，我们一般将其发布到公司内部的软件仓库或者以开源形式发布到 Maven 的中央仓库（Maven Central Repository），下面我们就以 spring-boot-starter-metrics 为例，向大家展示如何将类似 spring-boot-starter-metrics 这样的 Java 类库发布到自己公司内部搭建的 Nexus 服务器上。

首先，你要有一套已经搭建完成并运行的 Nexus 服务器（至于怎么做，问"百度"还是"谷歌"，你自己看着办），然后我们需要对 spring-boot-starter-metrics 的 pom.xml 附加一点儿发布相关的内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol</groupId>
    <artifactId>spring-boot-starter-metrics</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-starter-metrics</name>
    <description>auto configuration module for dropwizard metrics</
description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.0.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <distributionManagement>
        <repository>
            <id>deployment</id>
```

```
            <name>internal repository for releases</name>
            <url>http://{ 内部 nexus 服务器地址 }/nexus/content/repositories/
releases/</url>
        </repository>
        <snapshotRepository>
            <id>deployment</id>
            <name>internal repository for snapshots</name>
            <url>http://{ 内部 nexus 服务器地址 }/nexus/content/repositories/
snapshots/</url>
        </snapshotRepository>
    </distributionManagement>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.
sourceEncoding>
        <java.version>1.8</java.version>
        <metrics.version>3.1.2</metrics.version>
    </properties>

    <!-- 其他配置 -->

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>io.dropwizard.metrics</groupId>
            <artifactId>metrics-core</artifactId>
            <version>${metrics.version}</version>
        </dependency>
        <dependency>
            <groupId>io.dropwizard.metrics</groupId>
            <artifactId>metrics-annotation</artifactId>
            <version>${metrics.version}</version>
        </dependency>
        <dependency>
```

```
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.8.7</version>
        </dependency>
    </dependencies>

</project>
```

主要关注我们添加的 distributionManagement 相关内容，用于将我们的当前项目与内部的 Nexus 服务器进行关联，这样，就可以将当前项目不同阶段的成品（比如 SNAPSHOT 版本或者 RELEASE 版本）发布到特定的仓库路径下。

但是，只是在项目的 pom.xml 中添加 distributionManagement 相关内容还不够，对发布服务器的安全管控等敏感信息不能与 pom.xml 一同公开，所以，还需要在 ~/.m2/settings.xml 配置文件中添加 Nexus 服务器访问和认证信息：

```
<server>
     <id>deployment</id>
     <username>deployment</username>
     <password>********</password>
</server>
```

因为我们前面 distributionManagement 定义的 repository 和 snapshotRepository 的 id 都是 deployment，所以，这里的 server 的 id 也是匹配性地指定为 deployment，至于 username 和 password，则完全是我们内部的 nexus 服务器对应的安全认证用的用户名和密码啦。

> 注意 将内部 Nexus 服务器的认证信息放到 maven 的 settings.xml 中并非什么好的实践，纯粹是为了便利性而牺牲安全性，二者之间需要根据情况做出权衡，如果对安全性要求比较高的公司或者组织，最好将这些认证信息移除，并只在管控的范围内使用，比如将这些认证信息回收到发布和部署平台这一可控的小范围环境中，而所有开发人员使用的 settings.xml 属于“消毒”后无安全认证等敏感信息的版本。

当 pom.xml 中的 distributionManagement 以及 settings.xml 中对应的 server 设定都准备好之后，我们就可以直接 mvn deploy 将 spring-boot-starter-metrics 或者类似的 Java Library 项目发布到内部 Nexus 仓库了。

对于 Java 类库类型的项目来说，并无明确的部署过程，如果说有，也是存

在于可独立运行项目的开发过程中，比如使用 lib 目录或者结合 Ant “部署”为项目的依赖，或者直接享受 maven、gradle、sbt 等编译工具提供的“透明”的依赖部署过程。

5.2.2 基于 RPM 的发布与部署方式

部署的目标服务器从硬件到系统软件，一般情况下都应该是尽量相同，这与软件的标准化目的相同，一个是可以减少应对不同类型实体的复杂度，另一个就是标准化硬件和软件之后，就可以通过工具批量化以“边际成本递减”近乎为 0 的做法来提升效率，减少成本。

所以，对于大部分互联网公司来说，在硬件标准化的基础上，还会使操作系统尽量统一，比如大多数都是使用稳定性和可靠性经过长期验证过的 Red Hat CentOS 系统，而 CentOS 本身经过长期的沉淀也有一套自己的系统管理工具，比如像 YUM 或者 RPM 这样的系统包依赖管理器（Debian/Ubuntu 等 Linux 发行版也有对应的 deb 形式的包管理器）。

像 RPM 这样的包管理器对系统软件包的依赖和配置提供了很好的支持，如果我们的微服务等可独立执行实体要部署到像 CentOS 这样的目标环境中，使用 RPM 完成微服务的发布和部署，对于运维人员来说几乎就是无缝衔接的。

而且，对于 SpringBoot 微服务来说，单单一个可执行的 jar 包实际上是远远无法达到发布和部署要求的，如果只是发布一个可执行的 jar 包，那就意味着在部署阶段，运维要做更多的事情来弥补某些缺失，比如：

- ❑ 启动参数是否调整？
- ❑ 配置文件是否修改？
- ❑ 安装部署结构如何规范？
- ❑ 资源的对接和映射要不要做？

但是，如果我们能够将整个软件交付体系标准化和规范化，然后通过 RPM 这样的发布形式将这些标注和规范固化到发布包中，那么，整个部署过程就可以简化为一条命令，这其实也是使用 RPM 这种系统原生包管理工具完成交付部署的好处：自动化的 orcherstration[⊖]，减少不必要的人工干预中间环节。

⊖ 我也不知道 orcherstration 中文怎么翻译更好，大家只需要知道它表述的其实就是运维的集中协调的过程即可。

使用 RPM 发布 SpringBoot 微服务，我们简单将这一个过程划分为几步，如图 5-6 所示。

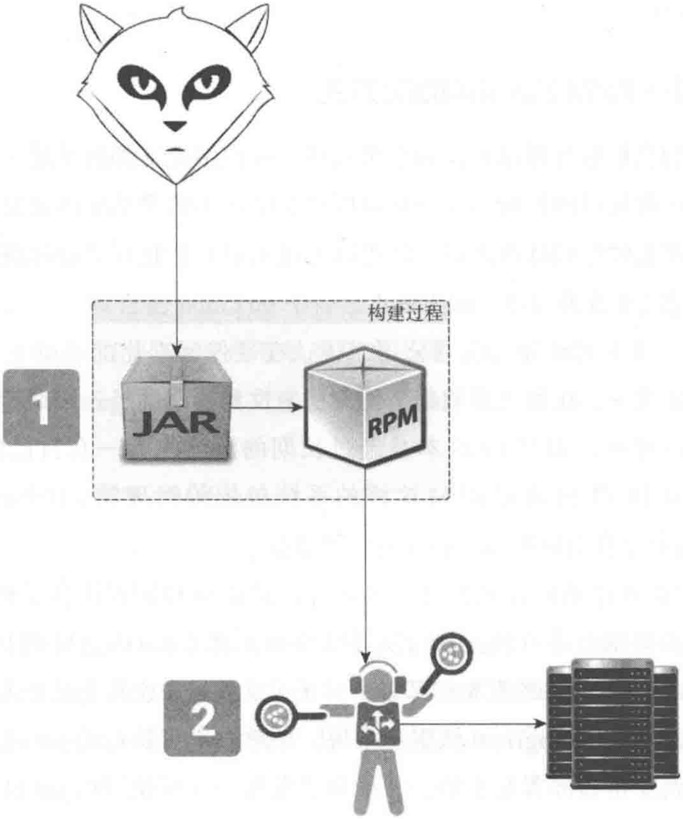


图 5-6 使用 RPM 交付的 SpringBoot 微服务发布流程图

首先，我们需要有一个特定的编译和项目构建环境，可以不管这个编译和项目构建环境是搭建在本地（比如你的开发机上），还是搭建在特定的一台服务器上，但这个编译和项目构建环境需要安装 rpmbuild，用来构建 rpm 包。

其次，我们不推荐 RPM 编译和构建的过程使用某些定义在项目编译脚本中的插件来完成，这样会让一些通用的逻辑散落在所有需要发布的项目中而不好治理（写代码姑且还要抽取通用逻辑到独立的方法或者类，更不要说这个粒度的关注点了）。所以，我们建议使用一个外部化的独立的编译服务器完成整个 SpringBoot 微服务的 RPM 发布和部署。

SpringBoot 微服务的 rpmbuild 脚本构建过程主要分几个主要步骤（如图

5-6 中 1 所标注):

1）调用标准的 mvn package 完成可执行 jar 包的打包。

2）根据软件交付规范，构建标准发布格式的 rpm 包：

- 使用 bin 目录存放根据脚本模板以及环境变量生成的启停脚本。
- 使用 config 目录（SpringBoot 默认文件系统中的配置目录名）或者 conf 目录存放特定的配置文件。
- 使用 docs 目录存放文档。
- 使用 agents 目录存放某些 javaagent。
- ……

3）在标准发布格式的基础上，生成从标准发布格式到具体目标环境的映射，比如原来应用的日志是默认打印到当前项目部署目录，而根据要求，我们希望打印到 /var/logs/{projectId}/ 或者其他服务器磁盘容量分配更大的分区，这个时候，可以在 rpmbuild 过程中指定安装类似的安装规则。

下面是一个简化的 SpringBoot 微服务的 rpmbuild 脚本定义：

```
Summary: metrics autoconfigure module for spring boot application
Name: spring-boot-starter-metrics
Version: {version}
Release: 1
Copyright: ...
Group: Applications/Productivity
Source: ...
URL: ...
Distribution:
Vendor: KEEp eVOLution, Inc.
Packager: Darren <afoo@keevol.com>

%description
%prep
%build
git clone .... # 检出代码到本地
cd {project folder}
mvn package
...

%install
%clean
%files
```

```
/{install_location}/{projectId}/bin/start.sh
/{install_location}/{projectId}/bin/stop.sh
/{install_location}/{projectId}/agents/jolokia-jvm-1.3.1-agent.jar
...
%attr(755, user, group) /{install_location}/{projectId}/agents/
jolokia-jvm-1.3.1-agent.jar
...
%doc
%changelog
```

然后需要调用 rpmbuild 的 spec 定义完成最终 rpm 包的构建：

```
rpmbuild -bb {projectId}.spec

# 然后 scp or sftp 生成的 rpm 包到指定的 rpm 仓库
```

我们可以像上面那样直接执行 rpmbuild 命令完成最终的 rpm 包构建，也可以将这些逻辑纳入编译构建脚本并部署到像 Jenkins 这样现成的持续集成服务器上，总之，执行完成后，打包好的 rpm 就发布到目标环境对应的 rpm 仓库了。

rpm 包发布到 rpm 仓库之后，就可以执行部署，比如通过 Salt 或者 ansible 在目标环境执行 rpm 或者 yum 命令，但具体的部署行为可能因为不同开发者的习惯和理念而有所不同。

有的开发者喜欢将不同目标环境的配置都一股脑地打包到发布包中，然后通过配置文件的命名和启动程序时单独指定一个环境变量来决定如何启用哪一个配置文件，对于这种做法，只需要打一个 rpm 包，同时也只需要搭建一个内部的 rpm 仓库，部署的时候，则需要运维人员根据具体的操作环境传递相应的环境变量来启停程序。

有的开发者则认为，一个软件实体发布的时候就应该是针对目标环境“装配”完备的，rpm 包中的各项配置都是针对特定目标环境配置好的，只要将 rpm 包部署到目标环境，就可以直接启动，启停完全无差别操作，唯一的差别是，rpm 包分别是根据目标环境发布和部署到不同的 rpm 仓库的，如图 5-7 所示。

以上两种策略并无优劣之分，但却有各自适合的场景：

1）在团队小，以人为本的时候，前者更适合，原因在于，整个软件交付链路更多是通过开发人员来协调和完成的。所以，开发、测试、运维一把抓，

即使是不同环境的配置文件，也都是为了开发人员方便，直接放到了项目目录下一起管理和修改。当到了线上，开发人员同时担当运维人员的角色，启停程序的可控性也很高，所以，可以在熟知自身程序的前提下，很好地完成整个链路的工作。

2）随着团队规模的扩张，职能更加明确，交付链路要承担的关注点也更多的时候，为了保证整个软件的交付质量，需要引入规范化的流程来关联和约束整个链路上各个环节和团队之间的工作。这个时候，开发人员的职责范围将缩小到明确的范围，测试团队、安全团队、应用运维团队等也将加入并根据流程各司其职，每个团队之间的工作需要横向关联的同时，又需要垂直隔离，这个时候，我们就需要从交付的源头一直到发布和部署，根据环境进行隔离，每个人即使只关注自己负责的事情，也可以让整个软件交付链路很好地工作，这考虑的是规范和流程对整体粒度上的把控和支撑。

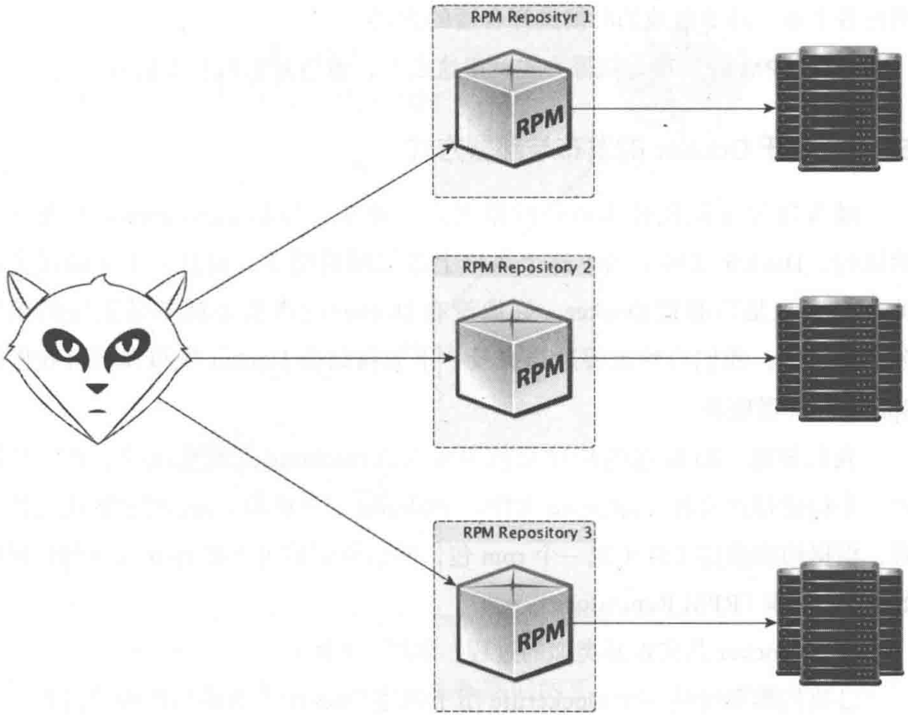


图 5-7　“根据不同交付目标环境，设置不同 RPM 仓库作为交付地点”示意图

对于小团队来说，微服务并不是什么太好的选择，高内聚的应用开发和部署单元，对整个交付链路的要求没那么高，也不需要更多自动化和平台化层面的投入和支持。

而一旦你选择了微服务的软件交付策略，数量庞大的微服务治理将耗费更多资源在支撑整个交付链路的自动化和平台化建设层面。否则，如此数量上的差异化的实体管理，单纯还靠人工拼苦劳是“捞不着好果子吃的”。所以，我们要标准化和规范化微服务的开发、交付、部署以及运维，从而收敛整条链路的治理复杂度，以近乎无差别的方式，完成各个环节上的工作。这个时候，规范、流程、平台是核心，对人的要求则适当降低。

说了这么多，其实就一点，如果团队要转向微服务的交付策略，那么，标准化、单一化的微服务发布和部署行为是大家努力的方向，虽然我们在说基于 RPM 这种特定的微服务发布和部署形式，但并不意味着我们只应该关注这一点或者这单一环节，只有系统的从整体的微服务交付链路和体系层面考虑，才能够在各个单一环节落地的时候选择合适的方案。

基于 RPM 的发布与部署方式就说这么多，希望对大家有所启发。

5.2.3 基于 Docker 的发布与部署方式

随着资源虚拟化技术的持续精进，一种基于容器（container）的方式开始风行，Docker 就属于当下这个风口上最耀眼的明星，而且，很多微服务相关的文章也是言必提 Docker，好像没有 Docker 的微服务就不是正宗的微服务了，所以，我们自然也要适当提及一下如何结合 Docker 发布和部署我们的 SpringBoot 微服务。

我们知道，RPM 包的构建是使用系统的 rpmbuild 工具完成的，该工具需要一个构建描述文件，即 .spec 文件，rpmbuild 工具读取 .spec 构建描述文件之后，根据构建描述文件生成一个 rpm 包，然后我们就可以把 rpm 包发布到相应的 RPM 仓库（RPM Repository）。

使用 Docker 其实也是类似的过程，如图 5-8 所示。

- 我们需要提供一个 Dockerfile 用于描述 Docker 发布成品的构建过程，这类似于 rpmbuild 需要的 .spec 文件。
- 在编写好要使用的 Dockerfile 之后，我们使用 docker build 命令读取

Dockerfile 开始构建一个 Docker 的 image，docker build 完成 rpmbuild 类似的功能，Docker 的 image 则类似于 rpm 包，即 Docker 的软件发布成品。

- 有了 docker image 之后，我们就可以将其发布到一个 Docker 的 image registry，这里的 image registry 就类似于 RPM 仓库（RPM Repository），而将 docker image 发布到 image registry 的过程，可以通过 docker push 完成。

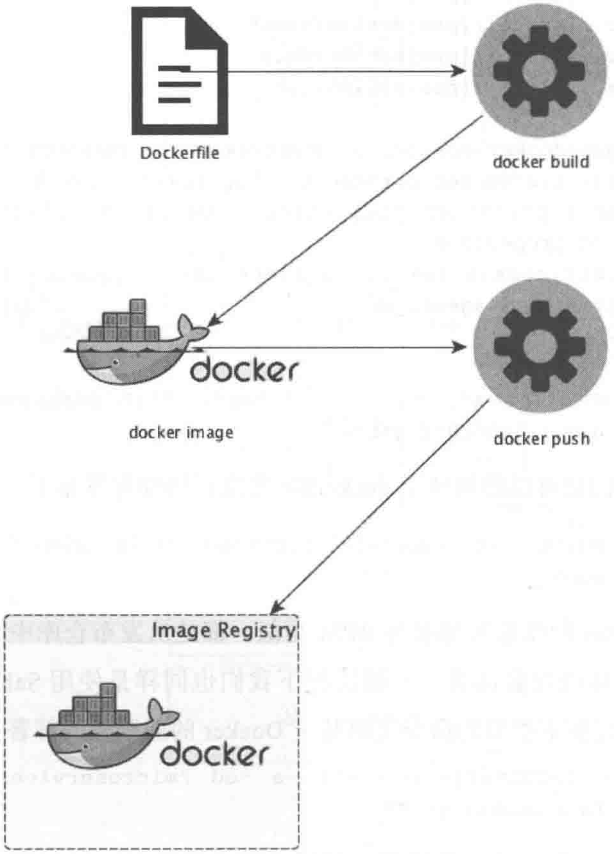


图 5-8　基于 Docker 的 SpringBoot 微服务发布流程图

所以，假设要以 Docker 的形式发布我们的汇率查询 SpringBoot 微服务，首先需要编写一个对应的 Dockerfile 来构建相应的 docker image：

```
FROM java:8
```

```
MAINTAINER AFOO <afoo@afoo.me>
LABEL groupId=...
LABEL artifactId=...
LABEL version=...
LABEL ...

USER deployer
EXPOSE 8080
ENV {key}={value}
...
VOLUME ...
RUN mkdir /{group}/{projectId}/config
RUN mkdir /{group}/{projectId}/agent
RUN mkdir /{group}/{projectId}/docs
RUN mkdir /{group}/{projectId}/lib

COPY target/docker-springboot-chapter4-0.0.1-SNAPSHOT.jar /{group}/
{projectId}/lib/docker-springboot-chapter4-0.0.1-SNAPSHOT.jar
COPY conf/application.properties /{group}/{projectId}/config/
application.properties
COPY agent/jolokia-jvm-1.3.3-agent.jar  /{group}/{projectId}/
jolokia-jvm-1.3.3-agent.jar
...

ENTRYPOINT ["java", "...",  "-jar", "lib/docker-springboot-
chapter4-0.0.1-SNAPSHOT.jar"]
```

之后，我们就可以使用这个 dockerfile 来进行构建并发布了：

```
$ docker build . -t "{groupId}/{artifactId}:{version}"
$ docker push
```

基于 Docker 的微服务部署与 RPM 类似，都是从发布仓库中拉取发布的成品，并在目标环境安装部署，一般情况下我们也同样是使用 Salt 或者 Ansible 之类的工具执行如下类似的命令完成基于 Docker 的微服务的部署：

```
$ ansible {cluster} -m shell -a "cd /microservices/{groupId}/
{artifactId}; docker pull"
```

为了简化基于 Docker 的发布和部署流程，实际上，以上演示的只是单人单微服务项目的方法，在讲究集团军作战的微服务场景下，我们希望的是能够快速、批量且标准化的形式完成数量巨大的微服务发布和部署，这就要求不能只盯着单一项目内部去思考如何实现发布和部署，而应该将视线从单一项目内部抽取出来，以更高的视角来审视如何快速地完成批量微服务的发布和部署。

笔者建议的一个思路是适当地弱化 Docker 属性，将发布和部署逻辑外部化到发布脚本中。外部化后的发布脚本将集中协调 Docker 基础设施，要发布的微服务上下文信息以及其他中间步骤，将微服务项目与 Docker 挂钩的唯一纽带也仅仅是一个模板化、标注化后的 Dockerfile，整个过程如图 5-9 所示。

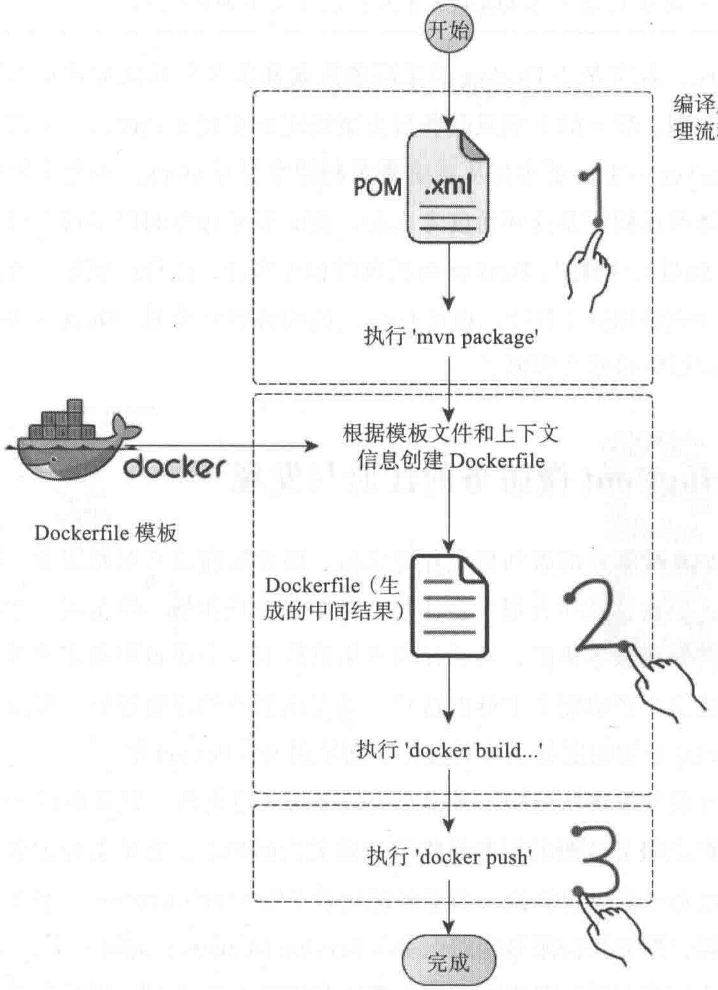


图 5-9　外部化的基于 Docker 的 SpringBoot 微服务发布脚步逻辑流程图

如此一来，对于所有希望以 Docker 形式发布的标准化的微服务来说，一套发布脚本即可完成所有微服务的发布和部署，而不需要每一个微服务自己去编写 Dockerfile 甚至发布脚本。

注意　上面的 Docker 实践并非 Docker 社区建议的最佳实践方式，这里更多只是为了简化说明和对比，Docker 背后是一套更为庞大完备的体系，比如 Docker 容器的注册和发现，容器的编排及调度等功能和系统，限于篇幅和内容定位，这里不再赘述。如果各位希望进一步深入了解 Docker，更多相关信息请参考 Docker 官网以及相关书籍和文档。

总的来说，其实基于 Docker 的微服务发布和部署与其他形式从本质上来说没有太大差别，唯一的差别只是各自方案特定的实现不同而已。大部分人选择和认同 Docker 方案，更多是从系统资源利用率以及 Docker 对整个软件交付链路的支撑体系比较完备这些角度考虑的，而微服务自身的很多特点以及需求（比如隔离、轻量），恰好与 Docker 能提供的相互吻合，或许这就是二者经常被“相提并论”的原因吧！（不过，对于 Java 应用和微服务来说，Docker 能给予的好处可能没有想象的那么多罢了。）

5.3 SpringBoot 微服务的注册与发现

SpringBoot 微服务部署到相应环境之后，即开始启动并对外服务。既然选择了微服务，那么就意味着很少会让某个微服务单兵作战，而是同一个微服务启动多个实例形成服务集群，然后让服务集群作为一个逻辑服务主体提供对外服务，而构建这个逻辑服务主体的过程，就是微服务的注册过程，服务的访问者如何访问到这个逻辑服务主体的过程，则是服务发现的过程。

服务的注册与发现从结构上来说是很简单的三角关系，如图 5-10 所示。

其中，提供服务注册的服务是整个三角关系的核心，它负责登记和保存哪些服务是可以对外提供服务的，当服务访问者（Service Accessors）想要访问某个服务的时候，需要先向服务注册服务（Service Registry）询问一下：“我想访问服务 A，请问现在都有哪些服务结点提供在线的 A 服务？”，然后服务注册服务就会返回服务 A 在线的服务结点，这样服务访问者就可以根据这些服务结点的信息直接访问相应的服务了。

SpringBoot 微服务的注册与发现由 SpringBoot 微服务提供服务的方式来决定，比如，如果我们使用的是基于 Dubbo 框架的微服务实现方式，那么，我们

的SpringBoot微服务可能采用的是基于Redis或者Zookeeper的注册与发现机制；如果我们提供的是Web API形式的微服务，那么，我们的SpringBoot微服务采用的则更多可能是基于DNS的服务注册与发现机制。

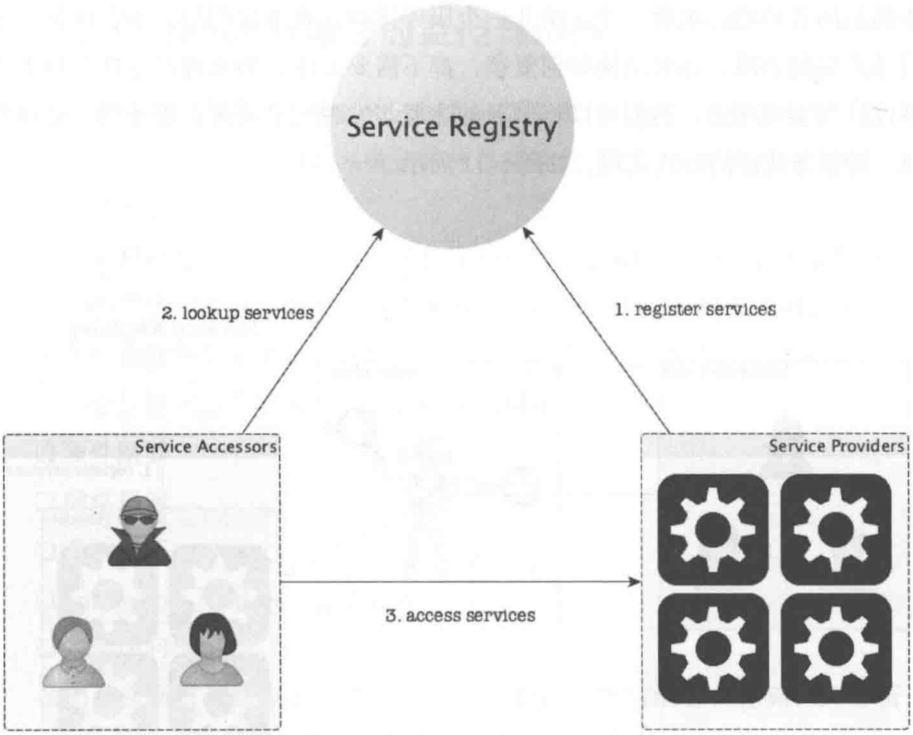


图5-10 服务注册与发现功能实现的典型结构图

虽然服务注册与发现的机制从原理上来说都是相同的，但具体实施的时候，方案上可以进行一些微调，这种微调主要以服务的发现者与服务的访问者角色和位置是否分离来区分。

一般情况下，服务的访问者同时也是服务的发现者，这通常以分布式系统的客户端为典型代表，比如HBase的客户端，以及Dubbo的客户端代理。这种模式的典型拓扑结构是，服务的访问者同时需要了解或者持有服务发现的逻辑，从而需要根据服务的发现逻辑首先去请求注册服务获得服务提供方的相应信息，然后再使用这些信息去访问服务提供者，这种模式很常见，但从服务访问者角度来看（实际上也是产品的用户角度），这不会带来最好的产品体验。

如果我们将提供的服务也看成一种（技术）产品，那么，一种好的产品设计应该是尽量减少用户的学习成本。也就是说，服务的访问者就是产品的使用者，他们不应该也不需要了解服务端有多少服务实例，这些服务实例又是使用什么方式实现注册的，注册之后又是通过什么方式发现这些实例并使用的，服务的访问者原始需求就一个，给我一个服务接口，我直接调用，不管你接口之后结点如何增减，部署结构如何复杂，都不需要关注，如果遵循这样的技术产品设计与实现理念，我们可以将服务的注册与发现完全屏蔽在服务端，这通常以一种服务代理的形式实现，如图 5-11 所示。

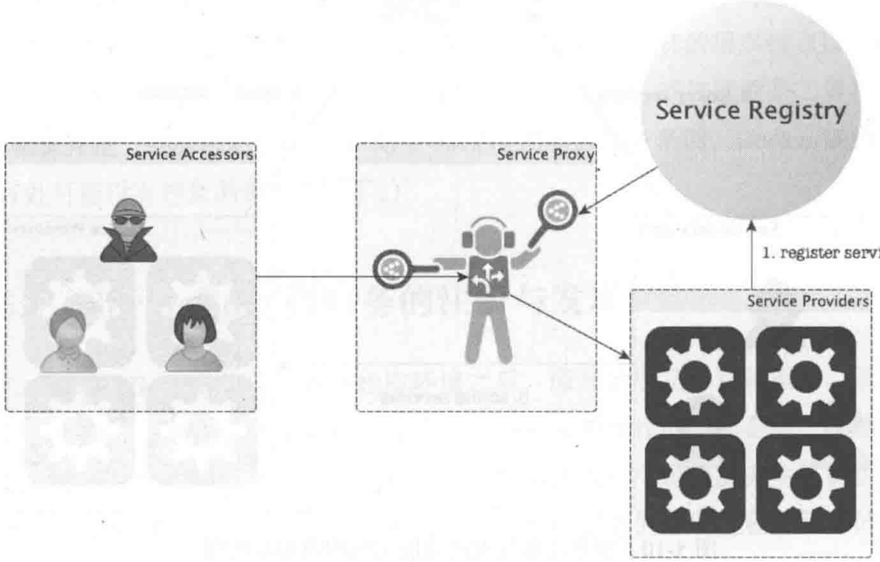


图 5-11　向用户屏蔽服务注册与发现实现细节的技术产品设计示意图

经过这样的微调之后，服务的访问者只需要与服务代理打交道，至于服务代理如何发现服务并路由服务的访问，对于服务的访问者来说都是透明的。

产品接口与产品实现要分得清楚，服务接口与服务实现也是同样的道理。原来将服务的访问与服务的发现都放在客户端的做法，我认为更多是一种偏工程师思维的实现方式，而如果我们稍微结合一些产品思维，我认为后者是一种更为合理的服务注册发现实践方式。从这个角度来看，在服务器端，HTTP 配合 DNS 或者 DNS-SD 或许是当下最有产品“范儿”的普适方案了，像 consul(https://www.consul.io/) 就是这种类型的服务注册与发现服务。

当然，SpringBoot 不会对任何一种服务注册与发现方案“挑食”，最终还是看大家选择的具体服务注册与发现方案是哪种，“路怎么走，你们看着办咯！”，但尽量不要将实现逻辑外溢到服务、系统或者产品之外。

5.4　SpringBoot 微服务的监控与运维

与大部分应用和系统一样，SpringBoot 微服务的开发、发布与部署只占其生命周期的一小部分，应用和系统运维才是重中之重。而运维过程中，监控工作更是占据重要位置。

运维的目的之一是为了保证系统的平稳运行，进而保障公司业务能持续对外服务，为了达到这一目的，我们需要对系统的状态进行持续地观测，以期望一有风吹草动就能发现并作出应对，监控作为一种手段，就是以此为生。

我们会从以下多个层面对 SpringBoot 微服务进行监控：

- 硬件层面
- 网络层面
- 系统层面
- SpringBoot 微服务的应用层面
- 服务访问层面

我们会从所有这些层面采集相应的状态数据，然后汇总，存储，并分析，一旦某项指标超出规定的阈值，则报警，在接收到报警通知之后，我们需要做出应对以改变现在系统状态不健康的局面，这一般通过预置的调控开关来调整应用状态，要么重启或者服务降级，也就是执行监控的“控”，整个过程如图 5-12 所示。

硬件、网络以及系统层面的监控，现有的一些监控系统和方案已经可以很好地提供支持，比如开源的 Zabbix 系统或者以报警为强项的 Nagios 系统，所以，本节不对这些层面的监控做过多介绍，我们将更多对 SpringBoot 微服务应用层面的监控进行实践方案的探索。

SpringBoot 微服务的内部状态，通过多种方式或者渠道可以知道：

- 打印的应用日志是一种 SpringBoot 微服务运行状态的反映形式。
- SpringBoot 微服务内部设置的一系列“传感器”可以为外界提供某些指

标的状态数据。

❑ SpringBoot 微服务内部追踪的一些 metrics 数据也是反映运行状态的一种方式。

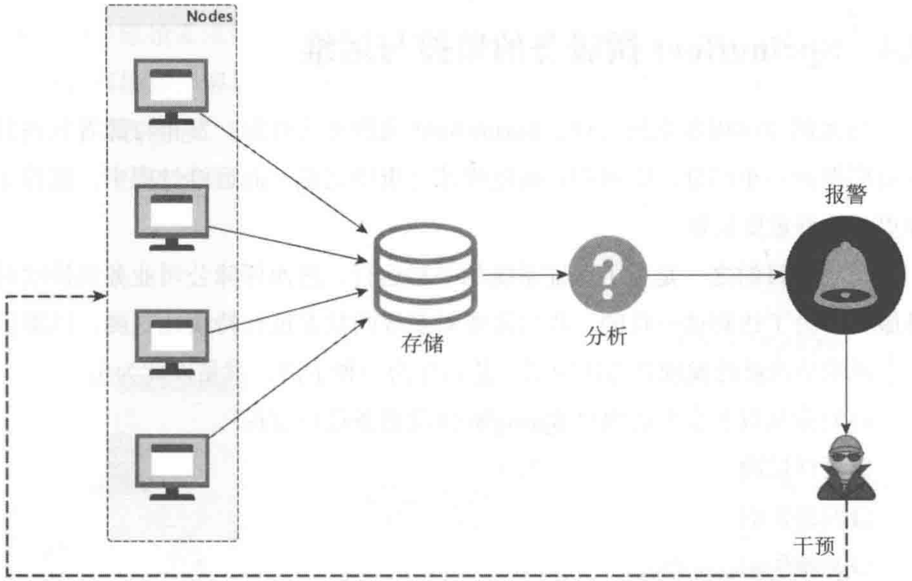


图 5-12　微服务基本监控体系示意图

任何可以反映 SpringBoot 微服务运行状态的数据，对于监控来说都是十分重要的“财产”，都应该尽量采集上来进行分析，从而在分析的基础上谋求对 SpringBoot 微服务的改进。

对于应用日志来说，在单机单结点的年代，我们只要登录应用部署的服务器，然后使用 tail -f 之类的命令就可以实时地查看应用日志信息，并决定如何做出应对。但对于 SpringBoot 微服务来说，数量上的特征已经决定了单机单结点的方法已经行不通了，如果还是一台台地去查看应用日志，我们不但会“疲于奔命”，而且还无法及时有效地发现微服务作为一个逻辑服务集群整体上的状态特征现在是什么样的，我们这时候需要的是一种集中式的日志采集、存储和分析平台。对于 Java 开发者来说，ELK 技术栈正是为此而生的（E = ElasticSearch, L = Logstash, K = Kibana），整个功能链路如图 5-13 所示。

不过，鉴于 ElasticSearch 对高频度的写入并没有很高的承受力，在正式的

生产环境中，我们一般会采用如图 5-14 所示的部署结构。

图 5-13 基本 ELK 技术栈功能链路示意图

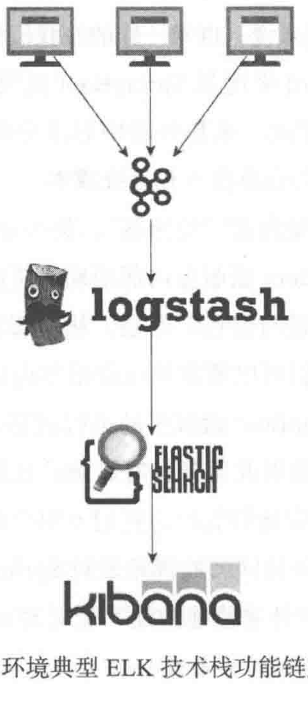


图 5-14 生产环境典型 ELK 技术栈功能链路部署示意图

即我们使用 Kafka 作为数据采集的缓冲区，以便减轻从大量应用结点采集日志并写入 ElasticSearch 的负担。

除了应用日志，应用内置的“传感器”数据以及 metrics 度量数据也都是比较重要的状态数据，对于 Java 程序来说（当然也包括 SpringBoot 微服务），JMX 是一项标准的监控手段。我们声称的应用内置“传感器”说白了就是一个个的 MBean。所以，我们要做的其实就是采集 SpringBoot 微服务的各种 MBean 的状态数据，然后分析或者报警。

要将集群中各个 SpringBoot 微服务的 MBean 状态数据集中采集上来，我们当然可以采用 JMX 默认的 JSR160 规范实现远程访问，而且像 Zabbix 这样的监控系统还默认提供了对 JMX 的访问支持。但是 鉴于 JMX 远程访问与防火墙之间的“纠葛”，以及 2015 年 Java 序列化漏洞暴露的不小危害，我更倾向于关闭 JMX 的远程访问，转而使用像 Jolokia(https://jolokia.org/) 这样的方案，Jolokia 的好处在于，它可以无侵入的方式（比如启动的时候或者启动之后挂载一个 javaagent）提供一种基于 HTTP 的 JMX MBean 访问通道（避免了防火墙穿透的问题），而且，它的 JMX 访问协议是统一的 JSON 格式，任何遵循 Jolokia JSON 协议规范的工具都可以对 MBean 的访问结果进行处理。

对于 metrics 度量数据来说，也是一样的。我们建议使用 dropwizard metrics (http://metrics.dropwizard.io) 来度量 SpringBoot 微服务的相应状态指标，然后通过 JMX + Jolokia 的方案统一采集并集中起来分析或者报警，从而避免多条处理链路上维护各自方案的资源投入和运维成本。

不管是应用日志，还是内置“传感器”，甚至 metrics 度量状态数据，这些反应的都是我们对 SpringBoot 微服务内部明确剖析后的感知，即以“白盒”的形式对 SpringBoot 微服务进行监控。但是，从内部看没问题，不意味着从外部看也没有问题，毕竟，我们可以覆盖并设置很多内部指标，但永远无法通过覆盖所有的指标来反映 SpringBoot 微服务的运行状态。而且，SpringBoot 微服务并非孤立的存在，它的周边对其服务也有影响，比如网络是否通畅。所以，要对 SpringBoot 微服务进行完备的监控，我们不但要从内部以“白盒”的形式进行监控，还需要从外部服务访问者的视角来对 SpringBoot 微服务进行模拟访问（即“黑盒”形式），从而内外兼修地构建一套针对 SpringBoot 微服务的应用监控体系。

实际上，SpringBoot 的 actuator 模块提供的健康检查（health check）功能就是一种允许以外部服务访问者的角度来监控微服务状态的现成方案。当然，更直接有效的方式则是直接发起对 SpringBoot 微服务的访问，从访问返回的结果来判断 SpringBoot 微服务整体上的运行状态是否良好。

5.4.1 推还是拉，这一直是个问题

我们发现，在各个监控层级各自的数据采集链路中，有些数据采集链路上采用数据客户端主动上报的方式（即 PUSH 方式），而有些则采用采集服务器端主动拉取的方式（即 PULL 方式）。

比如，基于 ELK 技术栈的应用日志采集采用的是基于 logstash agent 的主动推送上报应用日志到数据采集服务的方式，dropwizard metrics 库的 GraphiteReporter 或者 GangliaReporter 也是同样的道理。而基于 JMX 或者 JMX + Jolokia 则是需要数据采集服务器端采用定时拉取（PULL）的方式采集状态或者 metrics 度量数据。至于 Zabbix，则是主动模式和被动模式都支持。

那么，到底哪一种方式更好呢？

实际上，可能这两种方式并没有好坏之分，只是两种不同的理念而已。只不过这两种理念散落到各个链路的不同地方，就会造成不小的困扰。好在，二者之间可以一定程度上适配，从而一定程度上减轻“脑裂”的痛苦。比如，虽然 JMX 或者 JMX + Jolokia 使用采集服务器端主动拉取状态数据的方式最自然，但我们也可以在 JMX 应用端设置一个 agent，由这个 agent 来本地拉取数据并上报，从而将拉取模式（PULL）转变为推送上报模式（PUSH），进而统一各个链路的数据采集模式。

不过，在系统设计以及实现的时候，团队内部的理念最好统一，然后才能朝着一个方向使劲。其实，不管选择哪种模式，围绕这种模式构建系统和工具都能形成一套完备有效的方案，而一旦分散，则团队的技术生态体系中就会成长出实际上是完成同一目的，却要耗费双倍人力的方案。

前面我们以监控系统中服务于“监”目的的系统为例说明了两种理念下产出的不同方案，而秉承同样的两种理念，在服务于“控”目的的系统设计上，也会出现两种系统样貌。比如，如果我们采用监控平台主动推送消息（PUSH）的方式对目标集群中的各个 SpringBoot 微服务进行行为干预，那么就需要我们

的监控平台在主动推送消息这一功能上要有很强的并行执行能力，像 eBay 的 parallec(https://github.com/eBay/parallec) 系统就是遵循这样的理念设计并实现的。而如果我们采用 SpringBoot 微服务端自行拉取控制信息或者状态的方式来完成对 SpringBoot 微服务的控制行为，那么我们就会设计出像“配置中心”那样类似的共享状态的方案出来。

所以，推还是拉，其实一直是一个问题，这个问题不在于谁优谁劣，而在于二者无法互相理解。

5.4.2 从局部性触发式报警到系统性智能化报警

要为 SpringBoot 微服务构建一套行之有效的监控体系，报警的有效性在其中至关重要。

大部分情况下，监控体系中各个层次都各自为战，监控系统一旦接收到本层次关心的状态数据，并且发现超出了规定的阈值，就会即刻报警，如图 5-15 所示。

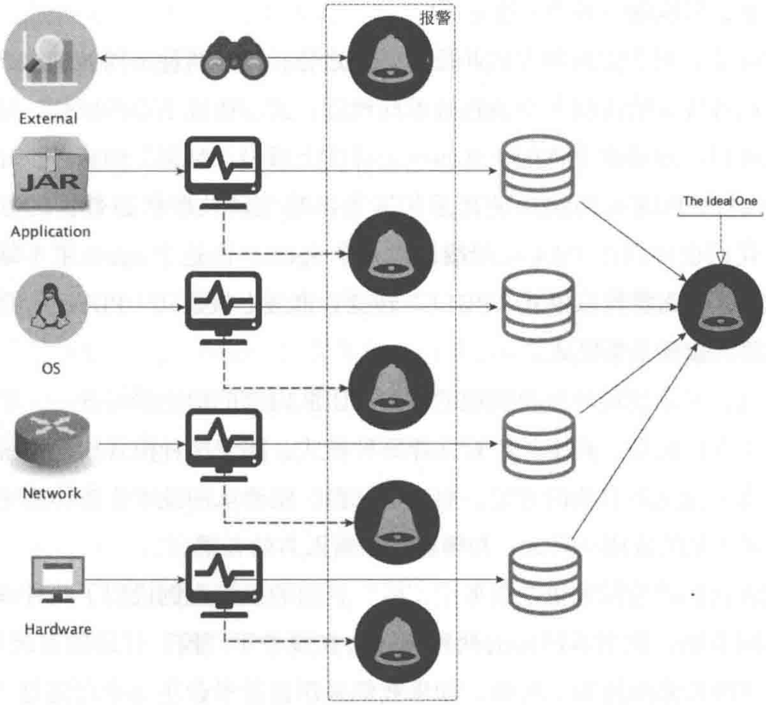


图 5-15 局部性触发式报警层次示意图

这种报警处理方式最大的问题就在于，它会生成很多的噪音：同一错误在多个 SpringBoot 微服务实例上报到监控系统之后，监控系统会无差别地悉数报警，而重复的报警往往会让运维或者应用负责人烦躁不已。

实际上，一种有效的报警体系应该是能够系统化地综合考虑所有监控层级的数据，然后汇总和分析，最终发出一个行之有效的报警。

我们称前一种为局部性触发式报警，而后一种叫系统性智能化报警。显然，系统性智能化报警应该是我们追求的最高目标。只有在条件有限的情况下，我们才会适当地容忍局部性触发式报警，最起码聊胜于无是不?

5.5　SpringBoot 微服务的安全与防护

一般的安全防护体系会从网络、系统和应用等多层次进行构建，以军事的上“深度防御”（Defence In Depth）为原则，层层设防，处处设岗，不过，几乎所有的 SpringBoot 微服务其实都是内网访问，所以网络安全层面我们可以适当降低防护要求。

因为笔者不是安全方面的专家，所以，本部分只对 SpringBoot 微服务在应用层面的安全防护做相应的实践探索。

SpringBoot 微服务开发上线的主要目的是对外服务，所以，安全起见，每一个 SpringBoot 微服务都需要鉴别哪些访问者是合法的，这就需要一个服务访问的认证和鉴权的过程（Authentication And Authorization），有了服务访问者的身份认证信息，我们的 SpringBoot 微服务才能够对其进行审计，进行访问限流，甚至直接拉到黑名单，所以，对 SpringBoot 微服务访问前端的安全防护是基本要求。

另外，虽然内网是相对安全的，但不排除某些人通过未知漏洞渗透到内网，然后嗅探内网通信获取敏感信息等情况发生，所以，对于微服务的访问者与提供方之间的通信信道，也可以进行适当加密。

当然，相对于 SpringBoot 微服务的后端管理接口的安全防护来说，SpringBoot 微服务访问的前端防护以及信道加密可以相对弱化一些，毕竟，SpringBoot 微服务存在的本身就是对外提供服务访问，况且还是相对安全的内网，但 SpringBoot 微服务的后端管理接口则不然，它们一旦被利用甚至滥用，

造成的影响可就无法估量了。

对于监控用的状态读取接口，除了某些敏感信息，随便怎么调用，这些状态读取接口都不会对 SpringBoot 微服务造成太大的影响（DDoS 除外）。但控制用的管理接口就危险多了，如果我们不加防护，任何一个人都可以任意调用微服务的 shutdown 操作。如果写个脚本批量关闭所有集群的 SpringBoot 微服务，那后果会怎么样？所以，即使我们弱化甚至省略 SpringBoot 微服务前端访问的安全防护，对后端的管理接口的安全防护一定不可省略！

我们要尽量从前到后为 SpringBoot 微服务构建一套相对完善的应用安全防护体系（如图 5-16 所示）：

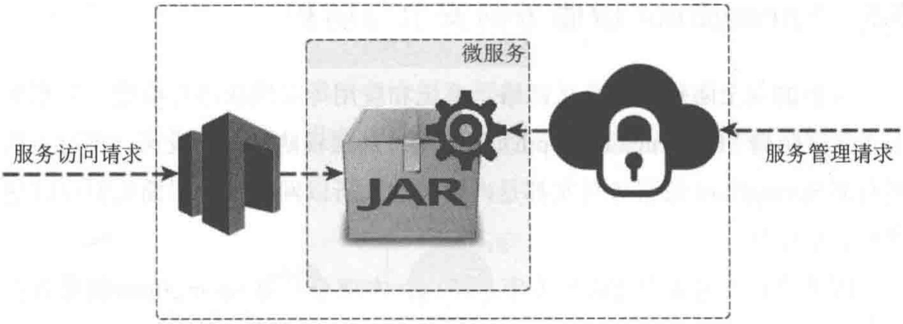


图 5-16　SpringBoot 微服务应用安全防护示意图

而且，后端管理接口的开放信道一定要跟前端访问接口的开放信道区分开。

如果使用 HTTP 协议，那么，不要用同一网络地址和端口；

如果使用 spring-boot-starter-security 进行安全防护，那么，也最好更改管理端口的地址和端口：

```
management.port=8888
management.address=127.0.0.1
```

如果使用 JMX + Jolokia 的方式开放了管理接口，那么建议启用 Jolokia 的 delegate 模式的 authMode：

```
java -javaagent:jolokia.jar=authMode=delegate,authUrl=https://
arthor.keevol.com ...
```

启用 authMode=delegate 之后，Jolokia 会通过指定的认证服务（authUrl）

以验证每次管理操作请求的合法性，从而保证最大限度地防护那些通过 Jolokia 开放的管理接口。

对于 SpringBoot 微服务的访问一般不会以个人为粒度进行授权，这不同于面向用户的应用程序，用户对 SpringBoot 微服务的访问，更多体现在通过其他应用程序对 SpringBoot 微服务的间接访问上。所以，对于 SpringBoot 微服务的访问授权我们一般以应用（Application）为粒度进行管理。

一般情况下，我们可以使用某些现有的 OAuth2 服务作为授权的管理和验证方，当然，我们也可以自己内部研发一套适合于 SpringBoot 微服务的认证与授权方案，比如我们前面提到的 Jolokia 的 authMode 使用 delegate 的时候，authUrl 后面指定的认证服务器，其实同样可以是一个 SpringBoot 微服务（比如叫 spring-boot-starter-arthor），它的主要作用就是：

- 提供应用访问的认证注册，比如访问者要以什么 Application 名称作为认证的主体。
- 为注册的 Application 提供相应的 access token。
- 每次访问请求需要认证的时候，服务提供方需要将请求提交给这个 SpringBoot 认证微服务进行验证，同时附带上 Application 标识和对应的 access token。
- 如果必要，请求的上下文信息也需要一并提交，然后进行授权检验。

当然，要为 SpringBoot 微服务构建一套完备的安全防护体系要做的事情还很多，留给各位去慢慢探索吧！

5.6 SpringBoot 微服务体系的脊梁：发布与部署平台

目前为止，我们分别从 SpringBoot 微服务的开发、发布、部署、运维监控、安全防护等多个层面进行了相应的实践探索。但实际上，这些实践探索暂时来看还都是简单、孤立的，单单强调任何一个都没有太多的意义，只有将它们关联起来形成一套 SpringBoot 微服务的支撑体系，才会发挥更大的作用，各自也会发掘出更多的附加值。这就跟一个手串一样，虽然每一个珠子都亮眼闪耀在外，但却需要一条平常不可见但实际存在的线将它们串起来，才能形成一幅拥有更大价值的手串。而这条线，其实就是我们需要关注和打造的支撑

SpringBoot 微服务体系的一套发布与部署平台。

要打造一套 SpringBoot 微服务体系的工作量完全不亚于打造一套传统的应用系统支撑体系，好在，SpringBoot 微服务是标准化的产物，所以，每一个环节都可以无缝地衔接（插头和插座标准相同）。所谓“闭门造车，出门合辙”就是这个道理，只要大家遵循一套标准和规范，各个环节纵使你闷头做事，出来的成果也可以标准化地接入更大的平台和生态圈。

但是标准化和规范化不能仅仅靠口头上的宣导，你无法保证所有时点所有人都能够覆盖到。所以，培训也好，会议也罢，只是软性的辅助手段来帮助规范和标准的实施，要行之有效地落实微服务体系的标准化和规范化思路，需要有硬性的手段来约束。虽不能武断地认为缺一不可，但一定是软硬结合才能达到最佳效果，而需要硬的时候，则是发布和部署平台作为实体系统发挥作用的时候。

另外，SpringBoot 微服务的交付链路上会涉及不同团队、不同角色的人员参与，如何通过流程将这些环节、角色、系统固化下来，以便即使数以万计的微服务重复高频的流过这些流程，依然能够保证交付的质量和效率，是发布和部署平台的核心职责之一。

支撑 SpringBoot 微服务交付的发布和部署平台作为“腰部力量”，就像鲤鱼要跳的那道龙门，具体的 SpringBoot 微服务就像那一尾尾的鲤鱼一样，跳过去了，你就别有一番洞天，在龙门（发布和部署成功）这边，你可以根据约定的规范和标准：

- ❑ 自动享受到系统监控层面的日志采集和分析支撑体系。
- ❑ 自动享受到为每一个 SpringBoot 微服务提供的资源分配，比如默认监控项和默认消息 topic 等。
- ❑ 自动完成服务的注册和发现。
- ❑ 自动享受到标准的安全防护。
- ❑ 自动享受到标准的 APM 平台支持。
- ❑ ……

发布与部署平台背后的一种可能的系统和功能统筹体系如图 5-17 所示。

SpringBoot 微服务发布部署和平台除了提供一整套的服务，还可以在发布之前或者发布期间对 SpringBoot 微服务做一些前置的管控，比如：

- ❑ 发布流程上，当前 SpringBoot 微服务是否完成了测试，是否完成了 SQL

审核，是否完成了安全评估。

- ❑ 规范的落实上，当前 SpringBoot 微服务是否符合日志规范（便于按照规范拆分字段进行更加细粒度的分析），是否启用了某些生产环境不该启用的自动配置模块，是否使用了过低版本或者存在漏洞的依赖。
- ❑ 资源分配上，当前 SpringBoot 微服务是否预先已经获得资源分配并录入 CMDB。

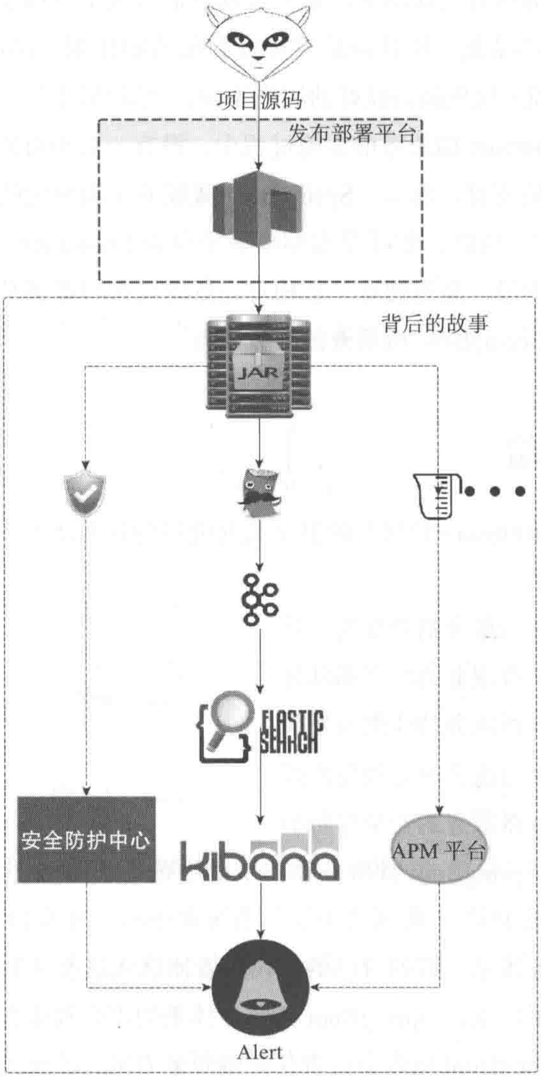


图 5-17　发布与部署平台的系统和功能统筹体系示意图

综上，一套SpringBoot微服务发布和部署平台，最少需要关注以下核心职能：

- ❑将公司的架构规范和标准以实体系统的形式固化和沉淀，软硬结合，方为始终。
- ❑将验证过的研发和软件交付流程落实到实体系统，统筹整条链路上的团队、角色和系统，完成SpringBoot微服务的高频、高质、高效的交付，“重复可复制的成功”，方为始终。
- ❑将公司内部所有关联的生产系统关联互通形成生态体系，注重发布和部署平台的产品化，使其向后发挥资源统筹的作用，向前为用户（开发、测试、运维）提供简洁良好的用户体验，“前轻后重”，方为始终。

如果在SpringBoot微服务的实施过程中，没有一套相对完备的发布和部署平台对其形成有效支撑，那么，SpringBoot微服务在组织内的实施将会“事倍功半”，有名无实。所以，称呼发布和部署平台为SpringBoot微服务体系的脊梁，一点儿都不为过。是否拥有一套相对完备的发布和部署平台，是一个组织是否已有效应用SpringBoot微服务的明显标志。

5.7 本章小结

我们围绕SpringBoot微服务的整个交互链路的各个环节，分别进行了相应的实践探索，包括：

- ❑SpringBoot微服务的开发与实践
- ❑SpringBoot微服务的发布和部署
- ❑SpringBoot微服务的注册与发现
- ❑SpringBoot微服务的运维与监控
- ❑SpringBoot微服务的安全与防护

但这也只是SpringBoot微服务浮在水面上的冰山一角，鉴于微服务的特点，我们要围绕它打造一系列的工具、系统和平台，才足以有效地支撑整个SpringBoot微服务体系。假如自己的公司或者团队无法投入有效的资源来打造一套符合自己组织的支撑SpringBoot微服务体系的平台和生态，那么，或许可以慎重地选择SpringBoot微服务或者任何微服务方案，传统的一站式开发和交付方式（Monolith）或许更加合适一些，性价比也更高一些。

Chapter 6

第 6 章 SpringBoot 与 Scala

Java 平台拥有三驾强力马车，即语言，类库和虚拟机，如图 6-1 所示。

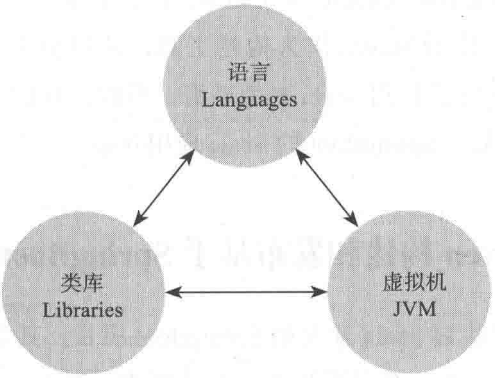


图 6-1　Java 平台“三驾马车”关系示意图

语言层面，Java 语言只是标配，像 Scala、Kotlin 等新生语言给了我们更多的选择，只要是可以跑在 Java 虚拟机上的语言，都可以享受到 Java 生态下的类库及框架等资源，SpringBoot 作为 Java 微服务框架，自然也可以为 Java 平台上其他语言带来微服务的暖风！

Scala 语言在业界的口碑相对比较复杂，但实际上，Scala 语言设计上的一些原则使得这门语言骨子里却是简单的：

- 统一性的设计（Consistency Principle），使得任何一个概念实体都可以被快速复制理解和使用“学习一次，多处使用”，比如 Collection 等对象的统一构建原则。
- 综合了过往计算机语言设计上的思辨，Scala 语言在设计上拨开表象，直指本质，为开发者免去了很多不必要的麻烦，比如类型后置和类型推导的设计，在业界设计静态语言层面已经是蔚然趋势。
- 融合而不是排斥，适当妥协而不是追求纯粹，可以抓住 OOP 和 FP 的各自优势，又能够从语言的易用性和实现的复杂性上做适当的权衡，这也是这门语言让我们如此兴奋的原因之一吧！

本章我们将以 Scala 语言为例，跟大家一起探索如何使用 Scala 语言和 SpringBoot 微框架来开发和交付相应的微服务。如果大家有自己更加心怡的基于 JVM 的语言，也希望大家可以举一反三，同时享受自己喜欢的语言和 SpringBoot 框架所带来的便利和乐趣。

要使用 Scala 语言开发 SpringBoot 服务，我们首先要决定的是使用什么构建工具，比如 Maven、Gradle 以及 Scala 生态下推荐的 SBT(Simple Build Tool)。相对来说，使用 Maven 作为构建工具，使得整个 SpringBoot 微服务的开发看起来从 Java 迁移到 Scala 更为平滑，所以，我们不妨先从探索使用 Maven 构建和发布基于 SpringBoot 的 Scala 应用开始。

6.1 使用 Maven 构建和发布基于 SpringBoot 的 Scala 应用

使用 Maven 来构建 Scala 开发的 SpringBoot 项目，其实与使用 Maven 构建 Java 开发的 SpringBoot 项目很接近，差异的地方很少，主要有几个地方需要改动：

1）因为 Scala 语言需要使用自己的编译器（Compiler）进行编译，所以，我们需要在 Maven 构建过程中使用一个编译 Scala 代码的 Maven 插件。

- Scala 项目在编译期间需要依赖 Scala 的 compiler 类库，所以，也需要将 org.scala-lang:scala-compiler 添加为 Maven 构建过程中使用的依赖。

2）Scala 应用运行期间需要依赖 org.scala-lang:scala-library，故此也同样需要加入到 Maven 项目依赖中。

以前面开发的汇率查询用的 Web API 项目为例，我们这次采用 Scala 来开发这个同样基于 SpringBoot 的 Web API 微服务。

首先通过 http://start.spring.io 构建 currency-webapi-with-scala "脚手架" 项目（Scaffolding），默认选择生成 Maven 项目（Maven Project），初始的 pom.xml 大致如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot.chapter5</groupId>
    <artifactId>currency-webapi-with-scala</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>currency-webapi-with-scala</name>
    <description>Demo project for Spring Boot and Scala with Maven
</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.1.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>com.keevol.springboot</groupId>
            <artifactId>currency-rates-service</artifactId>
```

```
            <version>1.0-SNAPSHOT</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

现在，我们要向这个pom.xml中添加“佐料”，添加的原则就是前面所说的那样，一个是Scala编译期的依赖，一个是Scala运行期的依赖，添加完“佐料”的pom.xml如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot.chapter5</groupId>
    <artifactId>currency-webapi-with-scala</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>currency-webapi-with-scala</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.1.RELEASE</version>
```

```
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.
sourceEncoding>
        <java.version>1.8</java.version>
        <scala.version>2.11.7</scala.version>
    </properties>

    <dependencies>

        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-library</artifactId>
            <version>${scala.version}</version>
        </dependency>
        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-compiler</artifactId>
            <version>${scala.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>com.keevol.springboot</groupId>
            <artifactId>currency-rates-service</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
```

```
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>net.alchim31.maven</groupId>
                <artifactId>scala-maven-plugin</artifactId>
                <version>3.2.2</version>
                <executions>
                    <execution>
                        <id>compile-scala</id>
                        <phase>compile</phase>
                        <goals>
                            <goal>add-source</goal>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>test-compile-scala</id>
                        <phase>test-compile</phase>
                        <goals>
                            <goal>add-source</goal>
                            <goal>testCompile</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <recompileMode>incremental</recompileMode>
                    <scalaVersion>${scala.version}</scalaVersion>
                    <args>
                        <arg>-deprecation</arg>
                    </args>
                    <jvmArgs>
                        <jvmArg>-Xms64m</jvmArg>
                        <jvmArg>-Xmx1024m</jvmArg>
                    </jvmArgs>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

其中，scala-maven-plugin 的各项自定义配置只是演示，大家可以根据实际情况进行相应配置项的调整，比如针对 Scala 编译器的输入参数，或者 jvmArgs 的配置。

按照约定，scala 代码一般都放在 src/main/scala 源代码目录下，既然 Maven 的构建描述符已准备好，我们就开始写 Scala 代码吧！

下面是 Web API 对应的 Controller 实现：

```
// ScalaCurrencyRateQueryController.scala 源码文件

@RestController
class ScalaCurrencyRateQueryController {
  @Autowired
  var currencyRateService: CurrencyRateService = _


  @RequestMapping(value = Array("/"), method = Array(RequestMethod.GET))
  def quote(symbol: String): WebApiResponse[ExchangeRate] = {
    val response: WebApiResponse[ExchangeRate] = new WebApiResponse
[ExchangeRate]
    response.setCode(WebApiResponse.SUCCESS_CODE)
    response.setData(currencyRateService.quote(CurrencyPair.from(symbol)))
    response
  }
}
```

完成了主体逻辑之后，剩下的就是与一般的 SpringBoot 微服务启动要做的事情一样，提供一个 Bootstrap 类：

```
// ScalaCurrencyWebApiApplication.scala 源码文件

@SpringBootApplication
class ScalaCurrencyWebApiApplication {
  @Bean
  def currencyRateService: CurrencyRateService = {
    val service: CurrencyRateServiceImpl = new CurrencyRateServiceImpl
    service.setRateRepository(new CurrencyRateRepository)
    service
  }
}


object ScalaCurrencyWebApiApplication {
  def main(args: Array[String]) {
     SpringApplication.run(classOf[ScalaCurrencyWebApiApplication],
args: _*)
  }
}
```

不同于 Java 将实例代码和静态代码都纳入同一个结构体，Scala 将这两种结构剥离为各自独立的实体，所以，main 启动类现在放在了 ScalaCurrencyWebApiApplication 的 companion object 中。

现在，只要通过 mvn package，然后 java -jar currency-webapi-with-scala-0.0.1-SNAPSHOT.jar 就可以顺利启动这个 Scala 开发的 SpringBoot 微服务了！

使用 Maven 构建 Scala 的 SpringBoot 微服务项目不单单只是开发期间过渡很平滑，其最大的好处更在于，原来围绕 Maven 打造的 SpringBoot 微框架的各项支持依然有效，比如 spring-boot-maven-plugin。而且，开发完的基于 Scala 的 SpringBoot 微服务可以无缝地衔接到我们自己的微服务交付链路中去（之前围绕着 Java 和 SpringBoot 打造的交付链路基础设施持续有效）。一旦通过微服务的发布和部署平台交付完成，基于 Scala 的 SpringBoot 微服务可以享受基于 Java 的 SpringBoot 微服务同样的待遇，从服务注册和发现，到监控与运维，甚至安全防护，岂不快哉？！

6.1.1 进一步简化基于 Maven 的 Scala 项目创建

使用 Maven 来构建 Scala 项目方便虽然方便，但依赖于每一个开发人员都去配置一遍项目的必要依赖，包括 Scala 插件的编译配置以及 Scala 的依赖类库，让每个人通过拷贝（Copy）之前的项目配置当然可以，但难免会出些纰漏。而从头开始所有相关配置项都配置一遍，又略显繁琐，那么，我们可不可以想办法来进一步简化类似的基于 Maven 的 Scala 项目创建和配置？

1. 使用代码片段管理工具

实际上，我们通常都会有收集的嗜好，尤其是作为一名软件开发者，都会有一套自己的代码片段管理工具或者套路，将一些常用的或者自己感觉经典的代码片段及配置内容进行摘录并保存。

现在有很多不错的代码片段管理工具，比如 Mac 系统上的 Dash (https://kapeli.com/dash)，它可以帮助我们一站式索引技术文档并管理代码片段，有了像 Dash 这样的工具，我们就可以将基于 Maven 的 Scala 项目配置文件整理成一个代码片段（Snippet）进行管理（如图 6-2 所示）。

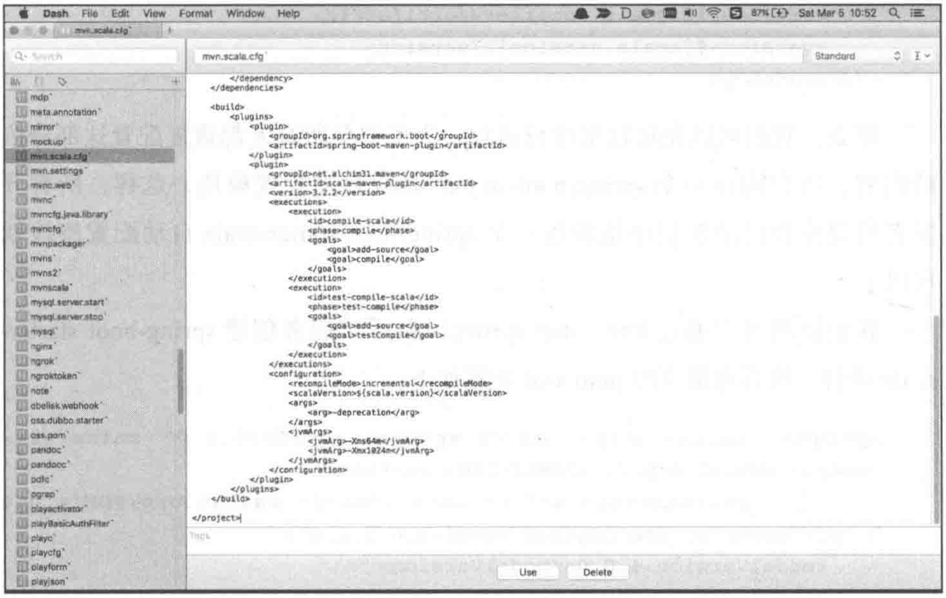


图 6-2 Dash 中管理的 Maven 配置相关的代码片段示意图

然后在使用的时候直接在 Maven 的 pom.xml 中输入该代码片段的标志信息：

```
mvn.scala.cfg`Dash
```

将会自动将我们配置的代码片段的内容展开并替换现有 pom.xml 的内容。

这种方法简单、快速、模板化、可重复，但只适合单一开发人员，如果一个团队或者组织内部所有人都使用这种方式，代码片段的内容同步，以及 Dash 等软件的授权等都是随之衍生出来需要进一步解决的问题。所以，为了能够使更大层面的开发者收益，我们需要寻找更加有效的方法。

2. 创建 spring-boot-starter-scala 简化 Scala 依赖配置

对于基于 Maven 的 Scala 项目来说，以下依赖是必不可少的：

```
<dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <version>${scala.version}</version>
</dependency>
<dependency>
    <groupId>org.scala-lang</groupId>
```

```
        <artifactId>scala-compiler</artifactId>
        <version>${scala.version}</version>
    </dependency>
```

那么，我们可以先从这里进行简化，为了避免所有人都重复配置这部分依赖内容，可以构建一个 spring-boot-starter-scala 自动配置模块，这样，所有开发者只要在自己的项目中依赖这一个 spring-boot-starter-scala 自动配置模块就可以了。

我们依然可以通过 http://start.spring.io 脚手架服务创建 spring-boot-starter-scala 项目，然后将最终的 pom.xml 配置如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot</groupId>
    <artifactId>spring-boot-starter-scala</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-starter-scala</name>
    <url>http://maven.apache.org</url>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>1.8</java.version>
        <scala.version>2.11.7</scala.version>
        <scala.maven.version>3.2.2</scala.maven.version>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>net.alchim31.maven</groupId>
                <artifactId>scala-maven-plugin</artifactId>
                <version>${scala.maven.version}</version>
                <executions>
                    <execution>
                        <id>compile-scala</id>
                        <phase>compile</phase>
                        <goals>
                            <goal>add-source</goal>
```

```
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>test-compile-scala</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>add-source</goal>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <recompileMode>incremental</recompileMode>
                <scalaVersion>${scala.version}</scalaVersion>
                <args>
                    <arg>-deprecation</arg>
                </args>
                <jvmArgs>
                    <jvmArg>-Xms64m</jvmArg>
                    <jvmArg>-Xmx1024m</jvmArg>
                </jvmArgs>
            </configuration>
        </plugin>

    </plugins>
</build>


<dependencies>
    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-library</artifactId>
        <version>${scala.version}</version>
    </dependency>
    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-compiler</artifactId>
        <version>${scala.version}</version>
    </dependency>
</dependencies>

</project>
```

然后通过 mvn install 或者 mvn deploy 将其发布。

现在，创建任何一个基于 Maven 的 Scala 项目时，pom.xml 配置内容就可以简化如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot</groupId>
    <artifactId>spring-boot-starter-scala-user</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-starter-scala-user</name>
    <url>http://maven.apache.org</url>

    <dependencies>
        <dependency>
            <groupId>com.keevol.springboot</groupId>
            <artifactId>spring-boot-starter-scala</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
</project>
```

是不是更加简洁和方便了？但是，等一下，如果大家都这样认为，那就有问题了。

实际上，这样的做法是行不通的，Maven 项目的依赖可以传递依赖（Transitive dependency），但是插件（Plugin）却不行。如果我们真得在前面的项目配置上进行工作的话，spring-boot-starter-scala-user 可以编译成功，可以打包成功，可以发布成功，但是，在整个过程中 Maven 无法识别 Scala 代码。

所以，如果要让基于 spring-boot-starter-scala 的想法可以实现，我们需要将 scala-maven-plugin 从 spring-boot-starter-scala 项目中移到 spring-boot-starter-scala-user 项目。

spring-boot-starter-scala 项目的配置现如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot</groupId>
    <artifactId>spring-boot-starter-scala</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-starter-scala</name>
    <url>http://maven.apache.org</url>

    <properties>
        <scala.version>2.11.7</scala.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-library</artifactId>
            <version>${scala.version}</version>
        </dependency>
        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-compiler</artifactId>
            <version>${scala.version}</version>
        </dependency>
    </dependencies>

</project>
```

spring-boot-starter-scala-user 的项目配置则如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot</groupId>
    <artifactId>spring-boot-starter-scala-user</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-starter-scala-user</name>
    <url>http://maven.apache.org</url>
```

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
    <scala.version>2.11.7</scala.version>
    <scala.maven.version>3.2.2</scala.maven.version>
</properties>

<build>
    <plugins>
        <plugin>
            <groupId>net.alchim31.maven</groupId>
            <artifactId>scala-maven-plugin</artifactId>
            <version>${scala.maven.version}</version>
            <executions>
                <execution>
                    <id>compile-scala</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>add-source</goal>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>test-compile-scala</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>add-source</goal>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <recompileMode>incremental</recompileMode>
                <scalaVersion>${scala.version}</scalaVersion>
                <args>
                    <arg>-deprecation</arg>
                </args>
                <jvmArgs>
                    <jvmArg>-Xms64m</jvmArg>
                    <jvmArg>-Xmx1024m</jvmArg>
                </jvmArgs>
            </configuration>
        </plugin>
```

```
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>com.keevol.springboot</groupId>
            <artifactId>spring-boot-starter-scala</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
</project>
```

现在看来，spring-boot-starter-scala 只是帮助我们简化了依赖管理，但对于 Scala 编译插件相关的简化则于事无补，像 spring-boot-starter-scala-user 这样的一大批基于 Maven 的 Scala 项目还是要一遍一遍地配置 Scala 编译插件相关内容。

不过，如果我们暂时能够忍受每个项目自己配置 Scala 编译插件的话（个人、小团队一般可以甚至也愿意），可以先进一步简化依赖 spring-boot-starter-scala 项目的 SpringBoot 微服务项目的创建，各位是否还记得我们经常用的 http://start.spring.io ？它罗列了很多现有的 spring-boot-starter 自动配置模块供我们在创建 SpringBoot 项目的时候使用，如图 6-3 所示。

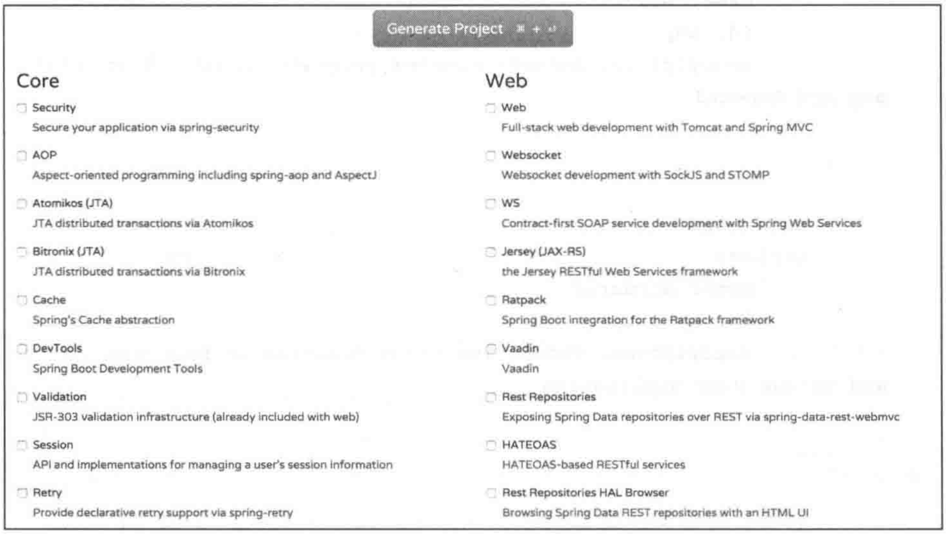


图 6-3 SPRING INITIALIZR 自动配置模块可用列表示意图

如果将我们的 spring-boot-starter-scala 也加入进去，大家在创建项目的时

候就可以不用配置 pom.xml 吗？

部署在 http://start.spring.io 的 SPRING INITIALIZR 项目服务实例我们是无权变更的，不过，我们可以在自己的公司或者组织内部搭建一个私有的 SPRING INITIALIZR 服务实例，笔者在前面也跟大家提到过，只要到 https://github.com/spring-io/initializr 获取项目然后部署到自己的服务，然后根据 https://github.com/spring-io/initializr#running-your-own-instance 的说明启动服务即可。

要将我们的 spring-boot-starter-scala 加入 SPRING INITIALIZR 的自动配置模块选择列表，需要先对自有的 SPRING INITIALIZR 实例进行一定的配置。

使用任何自己喜欢的文本编辑器打开 application.yml 配置文件（位于 {SPRING INITIALIZR 根目录 }/initializr-service/)，然后在 dependencies 部分的末尾添加如下配置内容：

```
dependencies:
  - name: Core
    content:
      - name: Security
        id: security
        description: Secure your application via spring-security
        weight: 100
      - name: AOP
        id: aop
        description: Aspect-oriented programming including spring-
aop and AspectJ

  ...

  - name: Ops
    content:
      - name: Actuator
        id: actuator
        description: Production ready features to help you monitor
and manage your application

  ...

  - name: Scala
    content:
      - name: Scala
        id: scala
        description: API documentation for the Actuator endpoints
```

```
repository: local-repo
groupId: com.wacai.springboot
artifactId: spring-boot-starter-scala
version: 0.0.1-SNAPSHOT
```

我们添加了一个名为 Scala 的展示配置区块，然后将 spring-boot-starter-scala 作为配置内容配置其中。

但这还不够，我们还需要告诉 SPRING INITIALIZR，当用户选择了 spring-boot-starter-scala 之后，在生成项目时，需要到哪里获取这个依赖，既然我们明确指定了 repository: local-repo，那么，就要配置一下这个名字为 local-repo 的仓库，让 SPRING INITIALIZR 知道这个 local-repo。

在 application.yml 配置文件中的 initializr -> env 部分添加一个 repositories 相关配置如下：

```
initializr:
  env:
    boms:
      vaadin-bom:
        groupId: com.vaadin
        artifactId: vaadin-bom
        version: 7.5.5
      cloud-bom:
        groupId: org.springframework.cloud

    ...

    repositories:
      local-repo:
        name: local-maven-repo
        url: file:///Users/fujohnwang/.m2
        snapshotsEnabled: true
```

我们添加了 local-repo 指向笔者工作机本地的 maven 仓库地址，大家也可以将自己公司或者组织内部私有 maven 仓库也添加进来。

提示　实际上，在配置 dependenies 的时候，并不一定要配置 repository: local-repo，这里纯粹是为了让大家了解这部分配置的更多信息。

关于 SPRING INITIALIZR 的更多配置详细说明，可以参考 https://github.com/spring-io/initializr/wiki/Configuration-format。

以上所有配置完备后，直接在 {SPRING INITIALIZR 根目录 }/initializr-service/ 下执行 spring run app.groovy 并最终启动 SPRING INITIALIZR，现在，访问 SPRING INITIALIZR 的服务地址就可以看到我们的最终成果了，如图 6-4 所示。

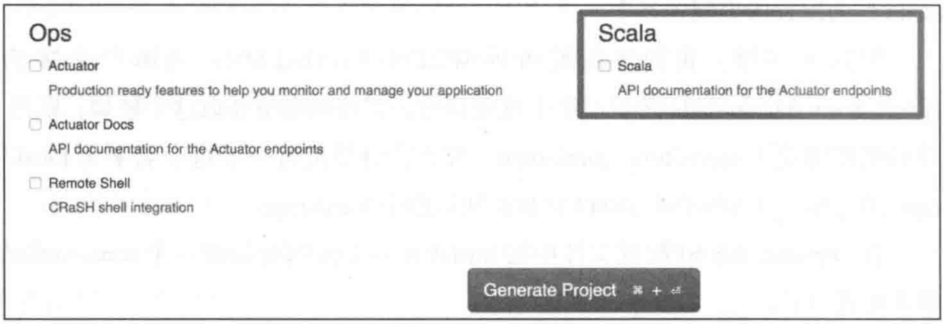
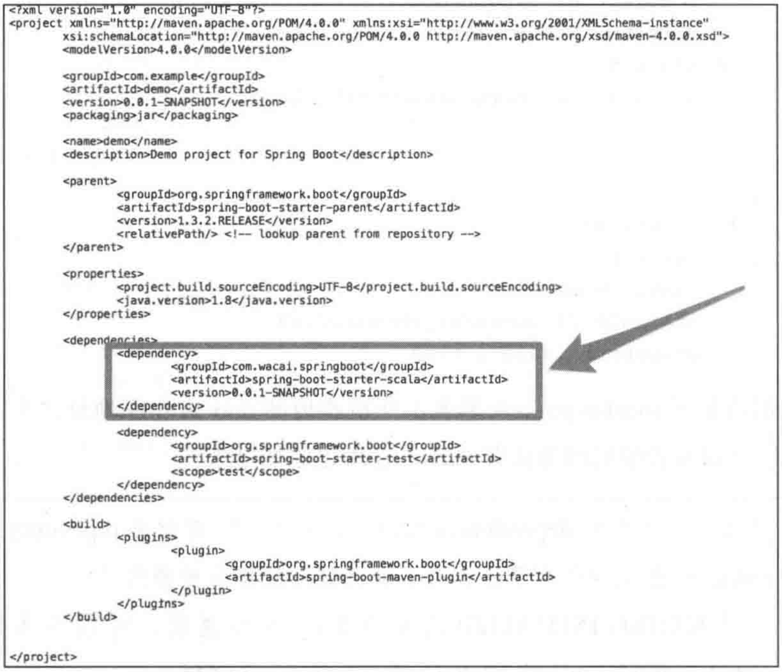


图 6-4　集成到 SPRING INITIALIZR 的 spring-boot-starter-scala 效果示意图

只要选择并单击生成项目之后，最终获得的项目配置中就已经自动获得了 spring-boot-starter-scala 的配置，如图 6-5 所示。



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>

        <groupId>com.example</groupId>
        <artifactId>demo</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <packaging>jar</packaging>

        <name>demo</name>
        <description>Demo project for Spring Boot</description>

        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>1.3.2.RELEASE</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>

        <properties>
                <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
                <java.version>1.8</java.version>
        </properties>

        <dependencies>
                <dependency>
                        <groupId>com.wacai.springboot</groupId>
                        <artifactId>spring-boot-starter-scala</artifactId>
                        <version>0.0.1-SNAPSHOT</version>
                </dependency>

                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                </plugins>
        </build>

</project>
```

图 6-5　生成的 Maven 项目配置中针对 spring-boot-starter-scala 的依赖配置示意图

关于基于 Maven 的 Scala 项目的依赖治理以及 spring-boot-starter-scala，我们暂时就介绍到这里。

虽然我们在依赖治理和工具的覆盖度、易用度层面做出了很好的探索，但依然没有简化 Scala 编译插件配置的繁琐操作，诸君还需与我继续努力。

3. 创建公司和组织级别的 Scala 项目的 parent pom

目前为止，Scala 相关的依赖管理已经比较令人满意了，现在，我们只要主攻如何简化 Scala 编译插件配置的简化就可以了。

好消息就是，Maven 的插件（Plugins）虽然不能通过组合的方式重用，但可以通过继承的方式搞定，所以，我们可以构建一个公司或者组织级别 Maven 项目的 parent pom 专门用于基于 Maven 的 Scala 项目。

下面是我们的示例项目，只有一个 pom.xml 定义：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.2.RELEASE</version>
        <relativePath/>
    </parent>

    <groupId>com.keevol.maven</groupId>
    <artifactId>scala-parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>pom</packaging>

    <name>scala-parent</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.
sourceEncoding>
        <java.version>1.8</java.version>
        <scala.version>2.11.7</scala.version>
        <scala.maven.version>3.2.2</scala.maven.version>
```

```
</properties>

<build>
    <plugins>
        <plugin>
            <groupId>net.alchim31.maven</groupId>
            <artifactId>scala-maven-plugin</artifactId>
            <version>${scala.maven.version}</version>
            <executions>
                <execution>
                    <id>compile-scala</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>add-source</goal>
                        <goal>compile</goal>
                    </goals>
                </execution>
                <execution>
                    <id>test-compile-scala</id>
                    <phase>test-compile</phase>
                    <goals>
                        <goal>add-source</goal>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <recompileMode>incremental</recompileMode>
                <scalaVersion>${scala.version}</scalaVersion>
                <args>
                    <arg>-deprecation</arg>
                </args>
                <jvmArgs>
                    <jvmArg>-Xms64m</jvmArg>
                    <jvmArg>-Xmx1024m</jvmArg>
                </jvmArgs>
            </configuration>
        </plugin>

    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>org.scala-lang</groupId>
```

```
            <artifactId>scala-library</artifactId>
            <version>${scala.version}</version>
        </dependency>
        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-compiler</artifactId>
            <version>${scala.version}</version>
        </dependency>
    </dependencies>
</project>
```

这个用作父项目（parent）的 maven 项目，其 pom.xml 的定义中将 Scala 编译插件（scala-maven-plugin）的配置以及 Scala 相关类库的依赖配置都囊括在内，这样，所有继承了这个父项目的子项目就都可以直接享受这些配置，而不用自己再去一一进行配置了，比如：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>com.keevol.maven</groupId>
        <artifactId>scala-parent</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

    <groupId>com.keevol.maven</groupId>
    <artifactId>scala-parent-user</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>scala-parent-user</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.
sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
```

```
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

注意 父项目的 packaging 类型为 pom，而不是一般的 jar 或者 war 等。

4. 使用 Maven Archetype 简化项目创建

不管是使用什么语言开发，创建项目都是一个永恒的主题，作为一线的开发工程师和领导者们往往也会为团队内部创建项目的千差万别而头疼，所以，脚手架（Scaffording）工具就成了研发体系中必要的一员。

如果我们紧盯和继续挖掘 Maven 的潜力，除了使用前面提到的那些方法，Maven 的 Archetype 功能也是一种可以帮助我们简化 Scala 项目创建的方案。

我们可以遵循 Maven 的 Archetype 规范创建一个 Scala 的模板项目（其实就是一个脚手架项目），然后将所有 Scala 项目相关的配置以及代码等都放到这个模板项目中，当每个开发者希望创建一个 Scala 项目时，只要使用这个 Scala 的 Archetype 项目创建一个模板项目出来就可以直接开发了。

创建 Maven Archetype 项目有两种方式：

- ❑ 根据 Maven Archetype 项目规范从零开始创建项目。
- ❑ 在一个现有的项目的基础上创建 Maven Archetype 项目。

鉴于我们已经有了好几个可以作为规范或者“模范”使用的 Maven 的 Scala 项目，从零开始创建一个 Archetype 就没有必要了，我们选择第二种方式，即在现有项目的基础上创建我们的 Maven Archetype 项目。

我们选择 currency-webapi-with-scala 项目作为模板项目用来创建 Archetype，然后再在生成的 Archetype 项目的基础上进行裁剪，最终获得一个理想的 Archetype。

首先，到 currency-webapi-with-scala 项目下执行 mvn archetype:create-from-project 命令，这将在项目的 target/generated-sources/archetype/ 目录下生成一个“草稿”版的 Archetype 项目，我们将在这个“草稿”版的 Archetype 项目基础上进行一定的裁剪。

target/generated-sources/archetype/ 下的内容结构是一个标准的 Maven 项目，其结构如图 6-6 所示。

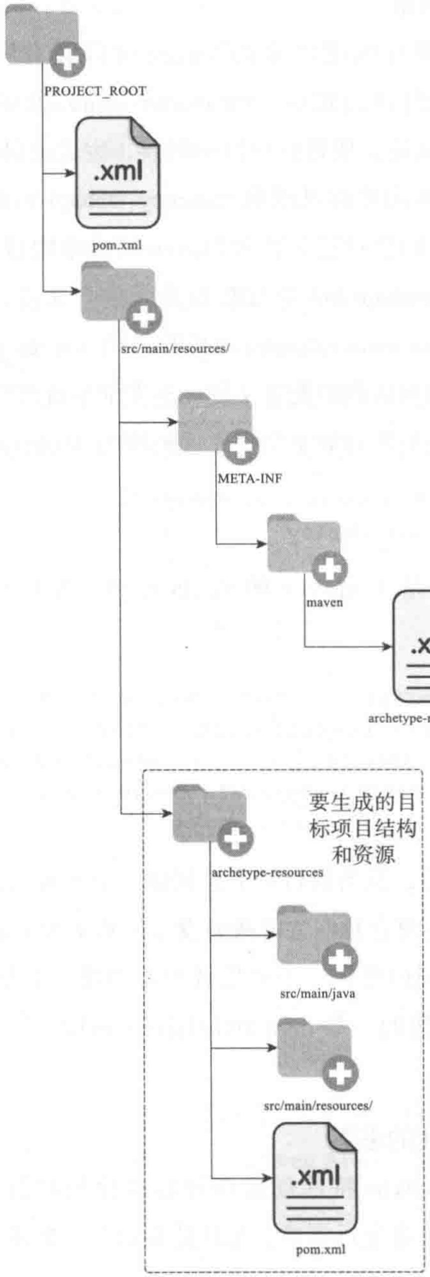


图 6-6 Maven archetype 项目结构示意图

我们可以对项目根目录下的pom.xml中的必要信息进行修改，比如groupId、artifactId、developers等，这些都是将来我们引用这个Archeteype项目创建项目时所关心的信息。

但使用Archetype项目创建出来的Maven项目会有哪些内容，则是由Archetype项目src下的内容决定的。src/main/resources/archetype-resources目录下的内容大体上认为就是生成目标项目的时候，生成的目标项目会拥有的东西，没做裁剪之前，这些内容都是根据currency-webapi-with-scala的内容"拷贝（Copy）"过来的，我们会对这个符合Maven项目结构目录下的内容进行裁剪，比如删除currency-webapi.iml等IDE相关元信息文件，添加.gitignore等必要元信息文件等。src/main/resources/META-INF/maven/archetype-metadata.xml文件是进一步细化裁剪规则的配置文件，更多细节就留给大家去探索吧！

以上裁剪完成后，我们执行如下命令完成最终的Archetype项目的发布：

```
$ cd target/generated-sources/archetype/
$ mvn install 或者 mvn deploy
```

现在，要创建一个基于Maven的Scala项目，我们只要执行如下命令即可：

```
mvn archetype:generate  -DarchetypeGroupId=com.keevol.springboot.
chapter5  -DarchetypeArtifactId=currency-webapi-with-scala-
archetype -DarchetypeVersion=0.0.1-SNAPSHOT -DgroupId=com.keevol.
springboot -DartifactId=new-scala-project-name -Dversion=0.0.1-
SNAPSHOT
```

以上命令执行完成后，在当前目录下会创建一个new-scala-project-name的目录，这就是一个完整的符合我们之前裁剪设定的基于Maven的Scala项目。

现在，团队或者公司内任何一个开发者想要创建一个基于Maven的Scala项目，只要执行以上类似的一条mvn archetype:generate命令就可以了，很简单，不是吗？

5. 畅想脚手架产品化的思路

使用Maven的Archetype特性创建各种标准化的项目是比较好的做法，但并非所有人都习惯使用命令行模式，尤其是有很多参数需要提供和配置的情况下。

对于使用 Intellj IDEA 的开发者来说，我们可以将自定义的 Maven Archetype 项目与 IDEA 进行集成，从而避免敲键盘和记住各种参数的痛苦，与 IDEA 集成后，IDEA 会提供一个标准的 Maven 项目创建对话框流程（Wizard），引导大家一步一步完成项目创建。

要将我们的自定义 Maven Archetype 项目集成到 IDEA，在 IDEA 的新建 Maven 项目窗口单击“Add Archetype”按钮进行自定义 Maven Archetype 的添加（需要首先选择“Create from archetype”才能够激活“Add Archetype”按钮），如图 6-7 所示。

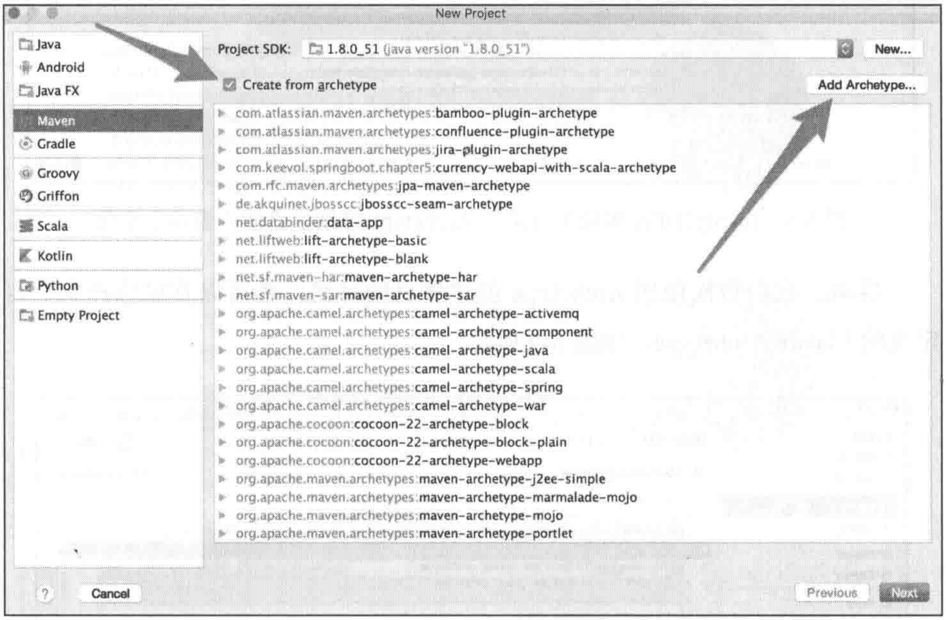


图 6-7　Intellj IDEA 中添加自定义 Archetype 示意图

然后 IDEA 会弹出新的对话框提示我们输入自定义 Maven Archetype 的相应信息，如图 6-8 所示。

根据我们的自定义 Maven Archetype 项目的 pom.xml 中的标识信息，完成相应项的输入并单击 OK 完成自定义 Maven Archetype 的集成：

```
<groupId>com.keevol.springboot.chapter5</groupId>
<artifactId>currency-webapi-with-scala-archetype</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>maven-archetype</packaging>
```

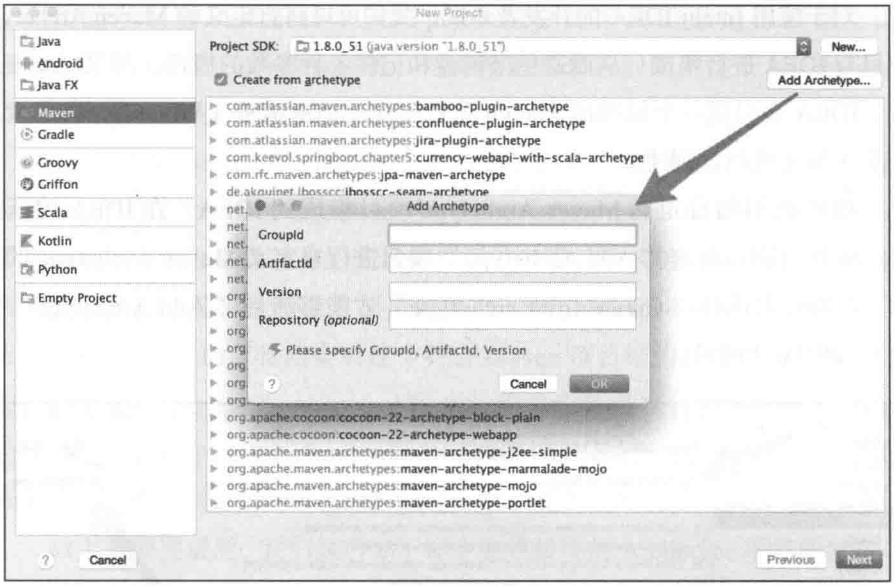


图 6-8　Intellj IDEA 中添加自定义 Archetype 详细信息输入窗口示意图

后面，我们每次使用 Archetype 创建项目的时候，就可以直接选择我们自定义的 Maven Archetype，如图 6-9 所示。

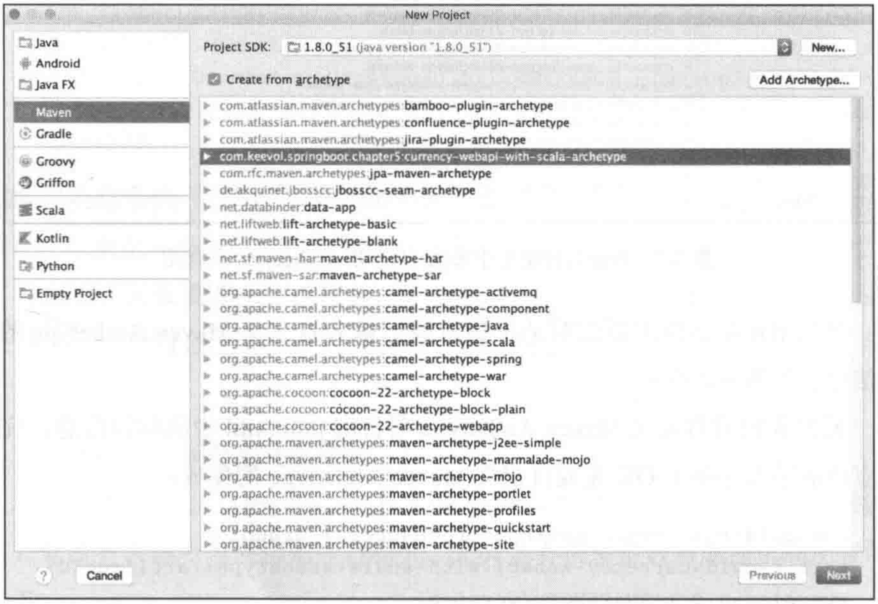


图 6-9　Intellj IDEA 中使用自定义 Archetype 创建 Maven 项目示意图

将Maven的Archetype与IDEA类似的IDE进行集成，固然可以将用户从命令行中解放出来（当然，前提是这些用户希望被解放），也重用了IDEA为类似场景提供的各项便利，但还是要求所有开发者都设定一遍，而且也并非所有开发者都使用IDEA甚至IDE，那么有没有更好的方法让所有开发者都收益？

SPRING INITIALIZR项目以及Typesafe[⊖]的Activator UI为我们提供了一种尝试思路。Web是一种统一部署且对用户环境无过多特定要求的产品形式，通过某个Web形式的脚手架产品，我们可以弥合并统一前面各种形式，如图6-10所示。

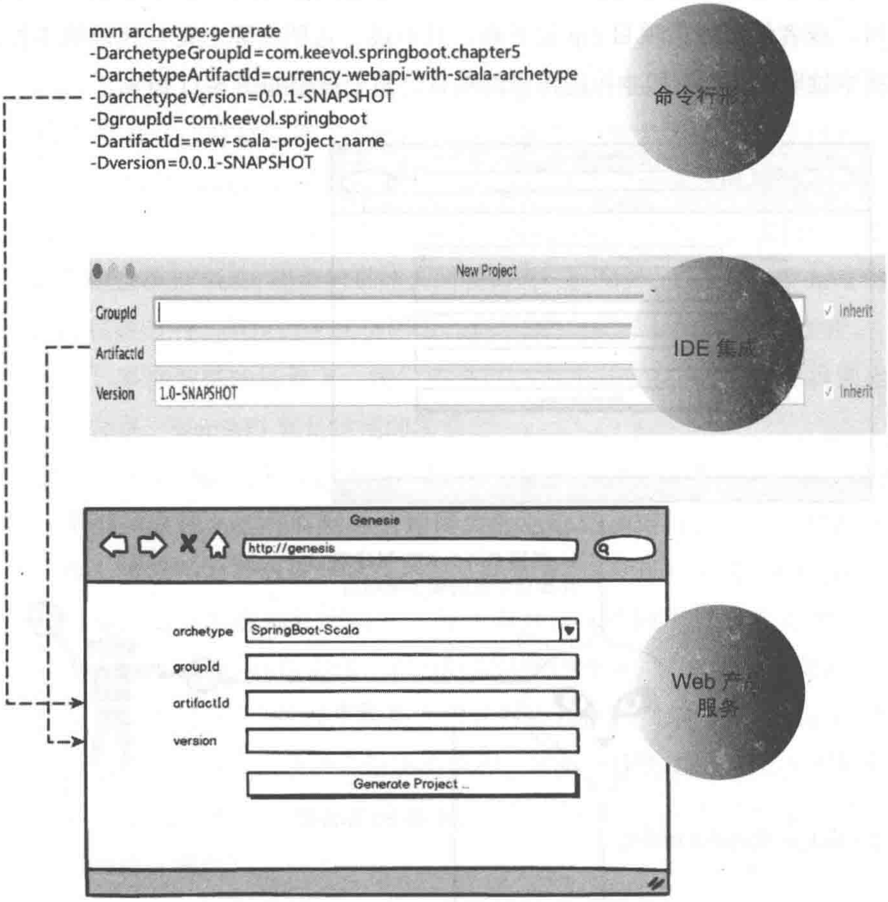


图6-10 使用统一的Web形式整合和提供脚手架功能示意图

⊖ 在写这部分内容时，Typesafe已经更名为Lightbend。

所有开发人员要创建新的项目，只要访问这个脚手架项目创建页面，然后选择项目类型，并输入必要的创建信息，就可以完成新项目的创建了。

SPRING INITIALIZR 在单击创建新项目之后，会为我们生成一个项目的 zip 包下载，作为开发者，我们要解压并提交到代码的版本控制系统，然后在其上进行工作，下载项目 zip 包的做法并非最优，其实我们还可以进一步改进。比如单击了生成项目之后，我们在服务器端生成了项目，但不是开启下载，而是直接在服务器端为开发者对接公司或者组织内的代码版本控制系统（比如 Gitlab），直接为开发者在代码版本控制系统中创建对应的项目，并把生成的项目内容提交到这个新建的项目下，告知用户直接到代码版本控制系统开始工作即可，或者依然提供项目 zip 包下载，只不过，这回实际上是从代码版本控制系统中拉取已经拥有版本控制信息的项目，整个过程如图 6-11 所示。

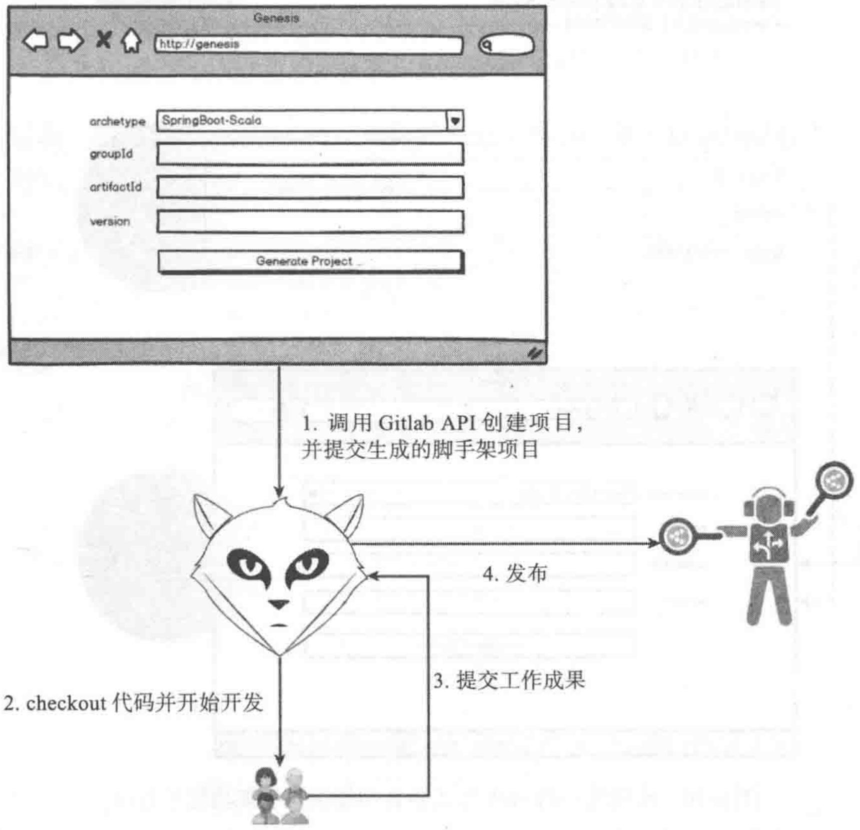


图 6-11　高度集成和体系化的脚手架功能与软件研发体系交互示意图

如此一来，开发者用户可以在任何一个端（比如计算机、平板电脑、手机等）任何时点创建模板化的项目，而不需要任何其他工具支持。

当然，像这样高度集成的产品化的研发体系支持并非所有公司或者企业愿意投入资源进行开发，各自都会在成本和效率之间寻求平衡，这里也只是跟大家一起畅想不同的思路，希望大家有所收获。

6.1.2 进一步简化基于Scala的Web API开发

虽然说使用Scala语言的语法来写SpringBoot微服务已经可以让Scala开发者们兴奋不已了，但说实话，这并没有很大程度上发挥二者各自的最大威力。

单向上来讲，从SpringBoot微框架出发，Java、Scala等Java虚拟机上的语言都会收益，这只是发挥了SpringBoot微框架的最大效能；但反过来，从Scala出发，使用SpringBoot微框架又会让Scala开发者受到一些束缚。比如像case class这样的Scala语言特性为了避免序列化兼容性等问题，我们一般不建议使用，但这确是Scala提供的最迷人的装备之一，另外，Scala依赖Java的各项生态是很融洽的，但反过来又会因为二进制兼容性等问题感觉如芒在背。

不过，某些特殊的场景下，我们还是可以为SpringBoot和Scala找到比较恰当的契合点，Web API就是这样的场景之一：

- 使用SpringBoot开发的Web API作为微服务独立部署，属于Endpoints类型的应用，这种类型的应用因为独立部署和运行，所以可以最大程度上容忍各种兼容性和复杂的“文化理念”，反正有冲突Endpoint内部解决，不会外溢到外面给别人带来麻烦；这不同于类库之类的依赖型（Dependencies）软件实体，它们更多偏组件性质，不能独立“存活”，需要“寄人篱下”，所以，如果它们有一堆自己的关联依赖，那么，就会带来更多兼容性以及冲突解决的负担。所以，作为Endpoint类型软件实体的Web API微服务，使用Scala是可以容忍Scala在依赖重和兼容性层面的“缺点”。
- Web API对外提供给用户的API接口是弱类型的，所以，我们在实现Web API期间即使使用各种强类型的对象和语言，其实对用户来说是无感知的，只要做好强类型数据对象和弱类型API之间的适配和转换，使

用 Scala 的各种语言特性必然无虞。

所以，基于以上两点考虑，我们可以在使用 SpringBoot 开发 Web API 的时候进一步深入使用 Scala 语言的一些特性。

鉴于我们已经实现了一个 spring-boot-starter-webapi 的自动配置模块用于简化 Web API 的开发，要进一步为 Scala 在此场景开辟捷径，只要在原有 spring-boot-starter-webapi 的基础上添加 Scala 特定的一些支持，就可以很大程度上简化基于 SpringBoot 和 Scala 的 Web API 开发。所以，我们决定创建 spring-boot-starter-webapi-scala 自动配置模块项目，该项目在享有原来 Web API 规范以及 API 文档的支持的同时，将进一步强化在使用 Scala 开发 Web API 的过程中使用 Scala 原生类型（比如 case class）作为数据对象的支持，尤其是 Scala 原生类型的序列化以及 Java 和 Scala 混合类型的序列化。

因为我们的 Web API 最终都是以 JSON 的形式来提供 API 的数据交互，所以，spring-boot-starter-webapi-scala 自动配置模块的主要一个新加功能就是加强 Scala 原生类型和混合类型的 JSON 序列化支持。基本上，我们会沿用 SpringBoot 在 Web 层 JSON 序列化默认方案上 Jackson 的选型，进一步依赖其 jackson-module-scala (https://github.com/FasterXML/jackson-module-scala) 来完成特定于 Scala 对象类型的 JSON 序列化工作。

依然基于 http://start.spring.io 创建我们的 spring-boot-starter-webapi-scala 项目，然后在原有脚手架项目的基础上进行配置裁剪和补足，最终完成项目的 pom.xml 内容类似于：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.keevol.springboot</groupId>
    <artifactId>spring-boot-starter-webapi-scala</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-starter-webapi-scala</name>
    <description>web api boot starter for scala</description>
```

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.3.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
    <scala.version>2.11.7</scala.version>
    <scala.maven.version>3.2.2</scala.maven.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>com.keevol.springboot</groupId>
        <artifactId>spring-boot-starter-webapi</artifactId>
        <version>1.0.0</version>
    </dependency>

    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-library</artifactId>
        <version>${scala.version}</version>
    </dependency>
    <dependency>
        <groupId>org.scala-lang</groupId>
        <artifactId>scala-compiler</artifactId>
        <version>${scala.version}</version>
    </dependency>

    <dependency>
        <groupId>com.fasterxml.jackson.module</groupId>
        <artifactId>jackson-module-scala_2.11</artifactId>
        <version>2.6.3</version>
    </dependency>

    <dependency>
```

```
            <groupId>com.fasterxml.jackson.module</groupId>
            <artifactId>jackson-module-paranamer</artifactId>
            <version>2.7.1</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

当前 pom.xml 中有几点内容需要大家重点关注：

- 我们添加了原来的 spring-boot-starter-webapi 自动配置模块依赖，目的是继续享有其对 Web API 规范以及 API 文档等功能的支持。
- 我们添加了针对 scala-library 和 scala-compiler 的依赖，毕竟这是为 Scala 项目服务的一个自动配置模块项目，这里配置了，所有使用 Scala 开发 Web API 的项目就不需要再逐一配置了。
- 我们添加了针对 jackson-module-scala 和 jackson-module-paranamer 的依赖，主要目的是强化针对 Scala 原生类型和混合数据类型到 JSON 序列化支持。

暂时我们不需要提供更多的功能支持，所以，只要通过 mvn install 或者 mvn deploy 将我们的 spring-boot-starter-webapi-scala 项目发布之后就可以供相关开发者使用了。

如果我们对原来实现汇率查询用的 Web API 进行重构，现在的 Maven 构建文件 pom.xml 可以直接简化成如下形式：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
<groupId>com.keevol.springboot.chapter5</groupId>
<artifactId>currency-webapi-with-scala</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>currency-webapi-with-scala</name>
<description>Demo project for Spring Boot</description>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.1.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
    <scala.version>2.11.7</scala.version>
</properties>

<dependencies>

    <dependency>
        <groupId>com.keevol.springboot</groupId>
        <artifactId>currency-rates-service</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>

    <dependency>
        <groupId>com.keevol.springboot</groupId>
        <artifactId>spring-boot-starter-webapi-scala</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
```

```
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>net.alchim31.maven</groupId>
                <artifactId>scala-maven-plugin</artifactId>
                <version>3.2.2</version>
                <executions>
                    <execution>
                        <id>compile-scala</id>
                        <phase>compile</phase>
                        <goals>
                            <goal>add-source</goal>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>test-compile-scala</id>
                        <phase>test-compile</phase>
                        <goals>
                            <goal>add-source</goal>
                            <goal>testCompile</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <recompileMode>incremental</recompileMode>
                    <scalaVersion>${scala.version}</scalaVersion>
                    <args>
                        <arg>-deprecation</arg>
                    </args>
                    <jvmArgs>
                        <jvmArg>-Xms64m</jvmArg>
                        <jvmArg>-Xmx1024m</jvmArg>
                    </jvmArgs>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

我们现在只要配置针对 spring-boot-starter-webapi-scala 自动配置模块的依赖就可以了，当然，Scala 编译插件 scala-maven-plugin 这里是直接配置当前项

目，可以根据前面的思路进一步简化配置。

因为之前是使用 Java 来定义值对象（Value Object），所以，这次我们直接使用 Scala 的 case class 来完成同样目的：

```
package com.keevol.springboot.chapter5.vo

case class Quote(symbol: String, sell: BigDecimal, buy: BigDecimal)
```

然后为了演示混合类型到 JSON 序列化的目的，我们在原来的 API 基础上开放一个查询最近 n 条行情的 Web API 接口定义：

```
  ...

  @RequestMapping(value = Array("/list"), method = Array(RequestMethod.
GET), produces = Array(APPLICATION_JSON_VALUE))
   def listQuote(symbol: String, @RequestParam(required = false)
lastN: Int = 10): WebApiResponse[List[Quote]] = {
      // Just a demo, you should retrieve the quotes from your
counterparty serivces
    // as per the requested currenypair symbol
      val quotes = List(Quote("USD/JPY", BigDecimal("113.94"),
BigDecimal("113.96")),
      Quote("USD/JPY", BigDecimal("112.94"), BigDecimal("112.96")),
      Quote("USD/JPY", BigDecimal("111.94"), BigDecimal("111.96"))
    )
      val response:WebApiResponse[List[Quote]] = new Web-
ApiResponse[List[Quote]]
    response.setCode(WebApiResponse.SUCCESS_CODE)
    response.setData(quotes)
    response
  }

  ...
```

可以看到，我们的 Web API 返回结果混合了 Java 对象类型（WebApiResponse）以及 Scala 对象类型（List 和 Quote），当然，只是实例代码，我们只是构造了模拟的行情数据，实际情况下，大家需要根据上游外汇交易商提供的行情数据进行下发。

最后，为了序列化可以成功，我们需要将 com.fasterxml.jackson.module.scala.DefaultScalaModule 注入 SpringBoot 的应用容器中，从而完成 Scala 类型数据序列化的整个准备工作：

```
@SpringBootApplication
class ScalaCurrencyWebApiApplication {

  // 其他 Bean 定义 ...

  @Bean
  def jacksonModuleScala(): Module = DefaultScalaModule

}

object ScalaCurrencyWebApiApplication {
  def main(args: Array[String]) {
    SpringApplication.run(classOf[ScalaCurrencyWebApiApplication],
args: _*)
  }
}
```

通过 mvn spring-boot:run 启动我们的 Web API 微服务，访问 http://localhost:8080/list?symbol=USDJPY，即可得到如图 6-12 所示的结果。


```
{
    code: 0,
    error: null,
  - data: [
      - {
            symbol: "USD/JPY",
            sell: 113.94,
            buy: 113.96
        },
      - {
            symbol: "USD/JPY",
            sell: 112.94,
            buy: 112.96
        },
      - {
            symbol: "USD/JPY",
            sell: 111.94,
            buy: 111.96
        }
    ]
}
```


图 6-12　汇率查询效果示意图

6.2　使用 SBT 构建和发布基于 SpringBoot 的 Scala 应用

SBT(http://www.scala-sbt.org/) 是 Scala 生态圈里的经典构建工具，虽然很

多人觉得 SBT 很复杂，还戏称其为 SB Tool，但其全称确是 Simple Build Tool，实际上，很多产品（包括像 SBT 这样的工具和技术产品）只有一个打动用户的特性就够了，对笔者来说，SBT 的 Triggered Execution 这一个特性就已经让我对其喜爱有加了。有了 Triggered Execution 这一特性支持，就可以在一个屏幕上写代码，而在另一个屏幕（或者窗口）实时地获得编译结果反馈，如图 6-13 所示，参与感十足。

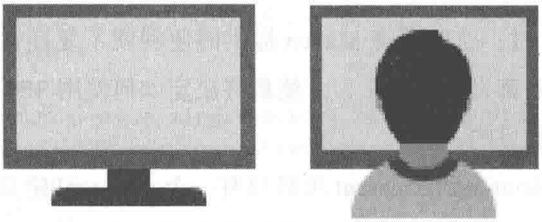


图 6-13 结合双屏使用 SBT 的工作场景示意图

当然，SBT 毕竟是一套新的工具体系，与 Maven 或者其他构建工具在配置和使用上还是会有不小的差异，这就意味着，如果我们要使用 SBT 来构建基于 Scala 的 SpringBoot 微服务项目，就需要针对 SBT 的特点做很多定制工作。

6.2.1 探索基于 SBT 的 SpringBoot 应用开发模式

SpringBoot 团队围绕 Maven 提供了很多便利和支持，比如我们只要使用 Maven 的继承特性，将 spring-boot-starter-parent 作为当前项目的 parent，就可以直接开发 SpringBoot 微服务项目：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.2.RELEASE</version>
</parent>
```

或者，如果当前要开发的 SpringBoot 微服务项目需要继承其他 parent 项目而不能使用 spring-boot-starter-parent，那么也可以使用 Maven 的依赖引入特性完成一系列 SpringBoot 相关依赖导入：

```
<dependencyManagement>
```

```
    <dependencies>
        <dependency>
            <!-- Import dependency management from Spring Boot -->
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>1.3.2.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

当使用 SBT 时，这些围绕 Maven 提供的便利就不复存在了，我们需要搞清楚这些便利背后都发生了什么，才能最终决定如何使用 SBT 进行相应的配置并最终达到相同的目的。

因为 spring-boot-starter-parent 项目只有一个 pom.xml 定义，且其 parent 就是 spring-boot-dependencies，所以，我们先沿着 spring-boot-dependencies 开始寻找端倪。

但是我们发现，spring-boot-dependencies 也只是一个只有 pom.xml 定义的“干瘪”项目，其 pom.xml 中仅仅通过 dependencyManagement 和 pluginManagement 固定了 SpringBoot 项目的一系列依赖和版本号，并无实际有用信息，所以，我们需要回头再从 spring-boot-starter-parent 的 pom.xml 中寻找线索。

不幸的是，spring-boot-starter-parent 也没有提供什么有用的线索，它的 pom.xml 中也是定义了相应的 pluginManagement 来规范依赖和依赖的版本，并在编译期间进行相应资源和配置的过滤。

到了这个地步，我们就只能另寻他途了。不过，既然大部分的 SpringBoot 项目都会从依赖相应的 starter 项目开始，那么，我们可以分析几个 spring-boot-starter-XXX 模块来看看它们有什么共性。假设我们关联查看了 spring-boot-starter-web、spring-boot-starter-jdbc、spring-boot-starter-actuator 等模块，就会发现，除了这些模块自身特定要提供的依赖，它们都同时依赖如下模块：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
```

而继续追踪下去到 spring-boot-starter 的 pom.xml 定义内部，最终才“柳暗花明又一村”：

```
...
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-autoconfigure</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-logging</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <exclusions>
            <exclusion>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.yaml</groupId>
        <artifactId>snakeyaml</artifactId>
        <scope>runtime</scope>
    </dependency>
</dependencies>
...
```

所以，一个 SpringBoot 项目得以成型，如下依赖基本是必备的：

- org.springframework.boot:spring-boot
- org.springframework.boot:spring-boot-autoconfigure
- org.springframework.boot:spring-boot-starter-logging
- org.springframework:spring-core
- org.yaml:snakeyaml

这些依赖其实也很好理解：

- 作为 SpringBoot 项目，spring-boot 依赖自然不可少。
- spring-boot-autoconfigure 依赖可以帮助我们进行自动配置，SpringBoot 项目的便利的核心就在于此。
- spring-boot-starter-logging 则配置默认的日志依赖。
- SpringBoot 项目就是一个 Spring 项目，所以 org.springframework:spring-core 也是必备依赖。
- org.yaml:snakeyaml，是运行期依赖，主要提供针对 yml 格式配置文件的支持。

至此，我们基本已经了解了 SpringBoot 项目背后的故事全貌，接下来就要介绍 SBT 了。

构建在 SPRING INITIALIZR 之上的 http://start.spring.io 可以在每次创建新的 SpringBoot 项目的时候提供“脚手架”功能，创建 SBT 项目其实也有这样的“脚手架”工具，即 Typesafe Activator(https://www.typesafe.com/activator/download)。

我们使用 Typesafe Activator 来构建一个新的 SBT 项目，用于构建同样功能的一个汇率查询 Web API 微服务，Typesafe Activator 安装完成后执行如下命令：

```
$ activator new currency-webapi-with-scala-sbt minimal-scala
```

其中，currency-webapi-with-scala-sbt 为我们要生成的 SBT 项目名，而 minimal-scala 为我们选择要使用的“脚手架”模板项目名。

初始生成的 SBT 项目的构建文件内容如下：

```
name := """currency-webapi-with-scala-sbt"""

version := "1.0"

scalaVersion := "2.11.7"

// Change this to another test framework if you prefer
libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.4" % "test"

// Uncomment to use Akka
//libraryDependencies += "com.typesafe.akka" %% "akka-actor" % "2.3.11"
```

我们需要对其进行定制和丰富一些内容：

```
organization := "com.keevol.springboot.chapter5"

name := """currency-webapi-with-scala-sbt"""

version := "1.0.0-SNAPSHOT"

scalaVersion := "2.11.7"

resolvers += "Local Maven Repository" at "file://"+Path.userHome.
absolutePath+"/.m2/repository"

libraryDependencies += "org.springframework.boot" % "spring-boot-
starter-web" % "1.3.2.RELEASE"

libraryDependencies += "com.keevol.springboot" % "currency-rates-
service" % "1.0-SNAPSHOT"

// Change this to another test framework if you prefer
libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.4" % "test"
```

因为我们已经了解了 SpringBoot 项目依赖的传递关系，所以，现在只要将 spring-boot-starter-web 作为核心依赖配置到 build.sbt 中，就可以信心满满地着手开发了（毕竟 spring-boot-starter-web 的依赖会一路上接力传递，必需的基础依赖一个都不会少）。除此之外，currency-rates-service 属于业务依赖，也需要一并遵循 SBT 的配置语法加以配置。因为实例项目是使用本地开发，所以，我们配置了相应的 resolvers 指引 SBT 从本地的 Maven 仓库中加载业务依赖，实际情况下，大家可以根据需要，将自己公司内部的 Maven 远程仓库或者其他 Maven 仓库也加入进来。

SBT 的项目源码结构与 Maven 的项目源码结构约定基本相同，所以，在 build.sbt 中所有配置完成后，就可以在 src/main/scala 目录下开始编写 scala 代码了：

```
// CurrencyRateQueryController.scala 源码文件

package com.keevol.springboot.chapter5.controllers

import com.keevol.springboot.chapter5.WebApiResponse
import com.keevol.springboot.services.currency.rates.CurrencyPair
import com.keevol.springboot.services.currency.rates.CurrencyRateService
```

```
import com.keevol.springboot.services.currency.rates.ExchangeRate
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.web.bind.annotation.{RequestMethod,
RequestMapping, RestController}

@RestController
class CurrencyRateQueryController{
  @Autowired
  var currencyRateService: CurrencyRateService = _


  @RequestMapping(value = Array("/"), method = Array(RequestMethod.GET))
  def quote(symbol: String): WebApiResponse[ExchangeRate] = {
    val response: WebApiResponse[ExchangeRate] = new WebApiResponse
[ExchangeRate]
    response.setCode(WebApiResponse.SUCCESS_CODE)
     response.setData(currencyRateService.quote(CurrencyPair.from
(symbol)))
    response
  }
}
// CurrencyWebApiBootStrap.scala 源码文件

package com.keevol.springboot.chapter5

import com.keevol.springboot.services.currency.rates.
CurrencyRateRepository
import com.keevol.springboot.services.currency.rates.
CurrencyRateService
import com.keevol.springboot.services.currency.rates.
CurrencyRateServiceImpl
import org.springframework.boot.SpringApplication
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.context.annotation.Bean

@SpringBootApplication
class CurrencyWebApiBootStrap{
  @Bean
  def currencyRateService: CurrencyRateService = {
    val service: CurrencyRateServiceImpl = new CurrencyRateServiceImpl
    service.setRateRepository(new CurrencyRateRepository)
    service
  }
}
```

```
object CurrencyWebApiBootStrap{
  def main(args: Array[String]) {
    SpringApplication.run(classOf[CurrencyWebApiBootStrap], args: _*)
  }
}
```

开发完成后，运行 sbt run 或者直接通过 sbt "run-main com.keevol.springboot.chapter5.CurrencyWebApiBootStrap" 即可启动微服务。

6.2.2 探索基于 SBT 的 SpringBoot 应用发布策略

我们基于 SBT 和 Scala 的 SpringBoot 微服务开发完成后，需要发布出去才能发挥其价值，要完成这种类型的项目发布，有几种策略可以考虑。

一种策略是使用 SpringBoot 推荐的可执行 jar 包发布形式，SpringBoot 团队提供的 spring-boot-maven-plugin 默认对这种可执行 jar 包有很好的支持，但是，对于微服务来说，我们前面说了，不建议采用这种方式，除非零星个别的一些项目，不需要强大体系支撑，仅靠人工就可以满足或者忍受。不过，如果因为某些因素必须使用 SpringBoot 建议的可执行 jar 包发布形式，那么，可以参考 SpringBoot 官网提供的参考文档的附录 D 的内容，结合 spring-boot-maven-plugin 的源码和逻辑，并实现一个相应的 SBT 插件即可。

另一种策略是使用 SBT 生态圈中已经形成一定口碑或者说基本上已经是事实标准的方案，比如使用 sbt-native-packager (https://github.com/sbt/sbt-native-packager)，通过 sbt-native-packager 这个 SBT 插件，我们可以将当前 SBT 项目发布为 ZIP、TAR、RPM、DEB 甚至 docker 等形式，相对于自己开发一个 SBT 插件完成可执行 jar 包形式的发布，显然直接使用 sbt-native-packager 是更明智的做法。

针对我们基于 SBT 和 Scala 的汇率查询 SpringBoot 微服务，要使用 sbt-native-packager 进行发布，我们只要对项目的 build.sbt 添加少许发布指引就可以了：

```
import sbt._
import com.typesafe.sbt.packager.universal.UniversalPlugin.
autoImport._
import com.typesafe.sbt.packager.MappingsHelper._
```

```
lazy val root = project in file(".") enablePlugins(JavaAppPackaging)

organization := "com.keevol.springboot.chapter5"

name := """currency-webapi-with-scala-sbt"""

version := "1.0.0-SNAPSHOT"

scalaVersion := "2.11.7"

resolvers += "Local Maven Repository" at "file://"+Path.userHome.
absolutePath+"/.m2/repository"

libraryDependencies += "org.springframework.boot" % "spring-boot-
starter-web" % "1.3.2.RELEASE"

libraryDependencies += "com.keevol.springboot" % "currency-rates-
service" % "1.0-SNAPSHOT"

// Change this to another test framework if you prefer
libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.4" %
"test"

// 发布相关
mainClass in Compile := Some("com.keevol.springboot.chapter5.
CurrencyWebApiBootStrap")

mappings in Universal += file("LICENSE") -> "LICENSE"

mappings in Universal ++= directory("config")
```

我们首先通过 import 引入相应的依赖，然后声明当前项目以 Java 应用（JavaAppPackaging）的形式发布（不是 Scala 应用类型是因为编译完运行期都是 Java 字节码的形式），最后通过 mainClass 来指定启动类是哪一个，以及应该将哪些其他文件或者目录打入发布包之中。在执行发布命令之前，我们还有最后一步要做，就是将 sbt-native-packager 添加为当前项目的依赖，在 SBT 项目结构下，这需要我们在当前项目根目录下新建 project/plugins.sbt 文件，然后添加如下内容：

```
resolvers += "Typesafe repository" at "http://repo.typesafe.com/
typesafe/releases/"
```

```
resolvers += Resolver.url("fix-sbt-plugin-releases", url("https://
dl.bintray.com/sbt/sbt-plugin-releases"))(Resolver.ivyStylePatterns)

addMavenResolverPlugin

addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.0.0")
```

之后，build.sbt 才可以依赖到 sbt-native-packager 并且编译成功，现在，只要我们运行 sbt universal:packageBin 命令，就可以在 target/universal 目录下获得一个 zip 形式的发布安装包 currency-webapi-with-scala-sbt-1.0.0-SNAPSHOT.zip，安装包格式包含以下内容：

- 自动生成启动脚本的 bin 目录。
- 存放了项目所有依赖的 lib 目录。
- 以及在 build.sbt 中通过 mapping 添加的一系列希望打包的文件和目录。

当然，zip 不是微服务发布的理想形式，但基于 RPM 和 DEB 等发布包需要依赖复杂的本地环境准备，故此，这里为了示例的简化，使用 zip 发布形式来说明如何使用 sbt-native-packager，以轻松愉快的心情完成基于 SBT 和 Scala 的 SpringBoot 项目发布。

提示 关于如何准备 RPM、DEB 等发布包的编译环境，以及如何配置 sbt-native-packager 使之相应的将当前项目发布为 RPM、DEB 等形式，可以参考 sbt-native-packager 插件的网站文档（http://www.scala-sbt.org/sbt-native-packager/index.html）。

最后一种策略（或许还有）我认为是最适合微服务的终极发布策略，即外部化（Externalization）的发布策略。

无论是在 Maven 项目的 pom.xml 中定义和使用 spring-boot-maven-plugin，还是在 SBT 项目的构建配置中定义和使用 sbt-native-packager 插件，本质上都是一种将通用逻辑分散到一个个单独项目中的做法，如果只是小团队以及少量项目，这样做是没有太大问题的。而一旦规模膨胀（对于微服务来说，数量、发布频度等指标规模膨胀很正常），加上人员流动、时机错配等一系列的变化因素，这种通用逻辑最终很可能会在这些一个个离散的项目中被“玩坏”，造成发布的成品千差万别，完全背离标准化的微服务这一理想轨道，进而带来整体微

服务交付链路上的一系列不必要的负累。

所以，为了避免“该标准化的地方却使用个性化”所可能引发的一系列问题，我们应该将这些发布和部署逻辑从单一项目中剥离出来，即外部化掉（Externalize it），将这些逻辑以发布和部署平台的形式抽象和固化，从而将原来因为各种因素导致的不确定性最大化地转变成确定性。

发布和部署平台是外部化的最终成果，每个 SpringBoot 微服务项目，不管你使用的是什么语言开发，也不管你使用的什么构建工具，只要你想用并且能帮助你提高开发效率，在开发阶段尽管用即可。一旦项目要发布，直接将发布请求对接发布和部署平台。对于 SpringBoot 项目的开发者来说，发布和部署平台是访问接口（Interface）；而对于发布和部署平台的开发者来说，则是服务提供方。服务提供方可以灵活替换和升级：

- 如果你的 SpringBoot 项目是使用 Maven 作为构建工具，那么，发布和部署平台实现层会自动感知并构建发布。
- 如果你的 SpringBoot 项目是使用 SBT 作为构建工具，那么，发布和部署平台实现层也会自动感知并构建发布。
- 如果组织内部当前的微服务体系是基于 RPM 标准统一发布形式，那么，所有 SpringBoot 项目不管你使用什么构建工具，都将以 RPM 标准形式发布。
- 如果组织内部要实现 DevOps 体系的升级换代（比如 docker），那么，只要发布和部署平台实现层进行变更和升级即可，SpringBoot 项目的开发者无须感知，发布和部署平台起到了很好的抽象和防火墙效果。

所以，对于一个拥有成熟微服务体系的组织来说，将发布等通用逻辑外部化到发布和部署平台，应该是最合适，也是目前最终极的策略。

6.3 本章小结

Java 平台的生态圈里有很多语言，Java 自不必说，Scala、Clojure、Kotlin 等更是后起之秀。

Java 平台的生态圈里也有很多构建工具，除了 Maven、ANT、Gradle 以及本章提到的 SBT 等，也是不一而足。

但不管这个生态圈如何繁杂和欣欣向荣，它们所生存的土地却是同一片土地，即Java虚拟机，这片土地上生长出来的资源也基本都是可以共享的，SpringBoot就是在这一环境下生长出来的一颗临风玉树，只要你喜欢，都可以靠过来乘凉，在本章中，Scala开发者现身说法，为同一片土地上喜好其他语言和工具的开发者开辟了一条探索之路，希望大家可以轻装上阵，依迹前行。

Chapter 7

第 7 章 SpringBoot 总结与展望

至此，我们的 SpringBoot 微服务体系探索之旅即将结束，回顾全文，我们先了解了微服务的基本概念和背景，以及微服务带来的优势和挑战，然后沿着微服务和 Spring 框架的脉络，一步步摸清楚了作为 Java 生态下相对成熟的 SpringBoot 微框架内部的工作原理，及有哪些现有的功能支持和扩展之道。当然，最重要的是我们很清楚，虽然 SpringBoot 为我们开发微服务提供了很大的便利，也带来了效率的高效提升，但对于一套完整的微服务体系来说，单单开发效率的提升并无助于微服务交付链路的整体效能，我们要做的是选定核心脉络（即 SpringBoot 微框架）之后，围绕这个核心脉络持续地投入和打造一系列跨越微服务交付链路各个阶段的工具、系统、平台和产品，并通过规范和合理的架构规划将所有这些看得见的、看不见的实体融合在一起，从而最终形成一套繁荣完备的微服务体系。

基础设施建设是微服务体系得以发挥最大价值的基本条件之一，我们单单基于 Scala 语言的 SpringBoot 微服务开发就摸索了多种方案帮助提升开发期间的效率（即使简单的脚手架功能也需要思虑和实践良多），更不要说为了发布、部署、监控、注册、发现和安全等多个阶段的基础设施投入了。比如，假设公司和组织内部以 RPM 为发布主要形式，那么，我们肯定需要搭建一套类似如图 7-1 所示的 RPM 的发布和存储分发系统。

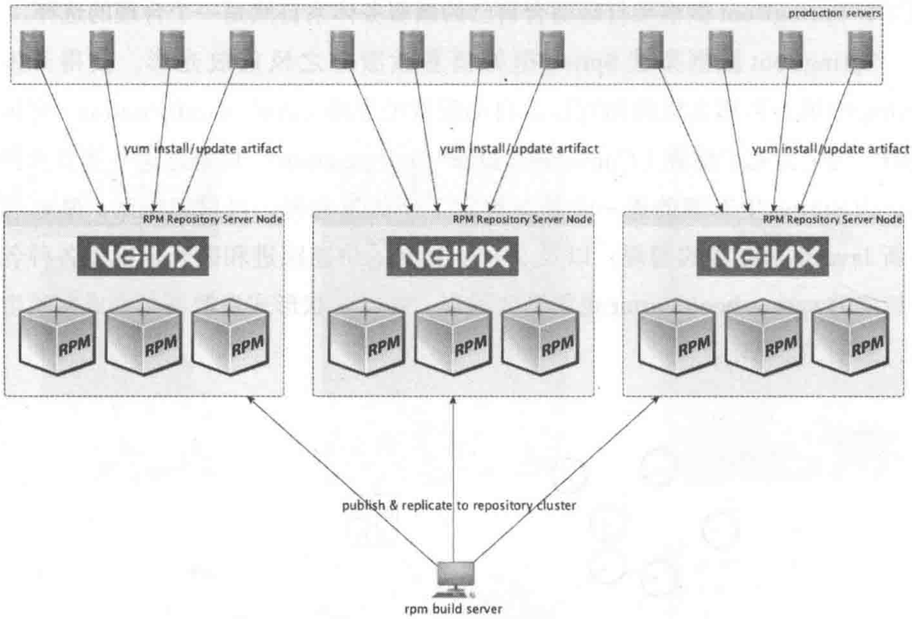


图 7-1 基于 RPM 仓库的软件交付和分发拓扑图

同理，如果要基于 docker，那么我们也肯定会内部搭建私有的 Image Registry，以及一系列服务注册和发现服务，服务编排和调度系统等。可以看到，要建设一套完美支撑微服务体系的基础设施，单是基础成本上的投入就不小，更不用说个人和组织能力的投入。因此，并非所有公司或者组织都适合引入并实施微服务战略。只有那些盈利模式强劲或者已经无生存压力的大中型机构，才会喊出“大中后台，小前台”的口号，因为他们有这个资源和能力完成基础设施建设并持续投入使之臻于完善。对于那些还挣扎在生死线上，业务探索方向还不清晰的中小公司来说，微服务战略即使有所耳闻，也最好脚踏实地，以务实的态度将其抛在脑后，以适合自己的方式野蛮生长并幸存下去！

即使一个公司或者组织有能力来推行微服务战略，但也不意味着你可以照搬其他成功实施了微服务战略的公司和组织的做法。实际上，即使每一家推行微服务战略的公司或者企业都会打造各自相应的微服务体系，但他们各自的微服务体系一定是不一样的，因为这需要结合公司现状和组织文化合理权衡，一家 Python 或者 Go 语言为主要开发语言的公司，其微服务体系一定是围绕 Python 或者 Go 语言生态构建的。而一家以 Java 为主要开发语言的公司，选择

围绕着 SpringBoot 微框架打造适合自己的微服务体系自然是一个合理的选择。

SpringBoot 微框架让 Spring 框架借着微服务之风重放光彩，使得开发 Spring 应用不再那么繁琐和冗长，自动配置的思路（Auto Configuration）将传统的“约定优先于配置（Conversion Over Configuration）”的理念进一步发扬传承，为 Spring 生态圈的进一步繁荣设定了新的里程碑，可以想象到，随着各种新 Java 技术方案的涌现，以及 Spring 社区的快速跟进和融合，更多各种各样相应的 spring-boot-starter 也将随之涌现，并以伞状形式发散甚至形成网状生态，如图 7-2 所示。

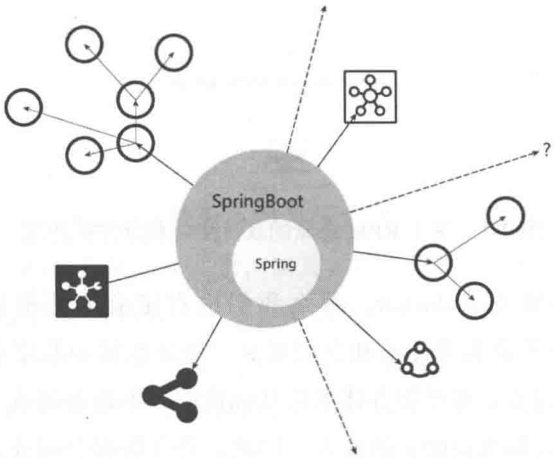
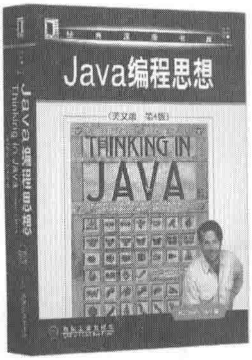


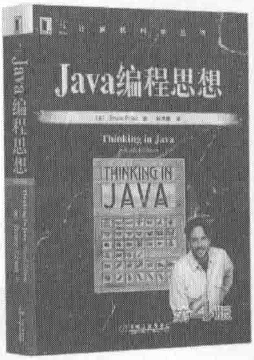
图 7-2　繁荣的 SpringBoot 生态展望示意图

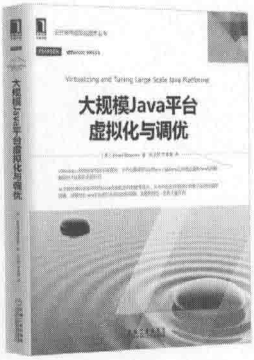
我敢说只要 Spring 框架存在，类似 SpringBoot 的实践也将变幻形式，不断涌现，因为只是需求的形式在变，但需求的本质没变，从来都是提高效率，提升生产率，让研发工作更方便高效罢了。如果哪一天 Spring 框架或者 SpringBoot 框架被颠覆，那也是被一种更加高效的方式所替代，如果有那么一天，那也是我们希望看到的，而且总会有那么一天！

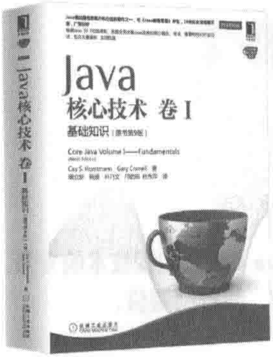
最后，希望大家能够享受这段 SpringBoot 微框架和微服务之旅，也希望大家可以有机会打造一套符合自身企业和组织特色的微服务体系。

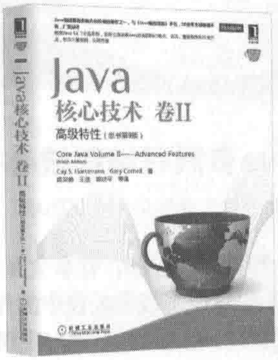
推荐阅读

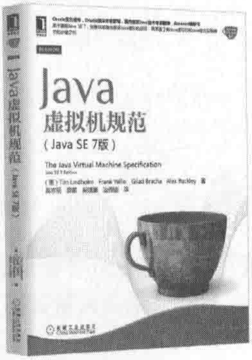

Java编程思想
（英文版 第4版）
THINKING IN JAVA

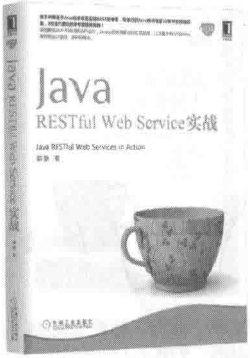

Java编程思想
Thinking in Java
THINKING IN JAVA

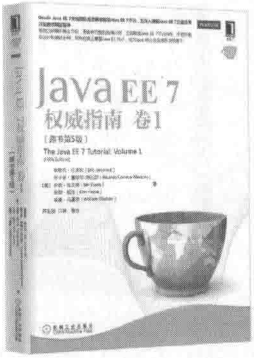

大规模Java平台
虚拟化与调优

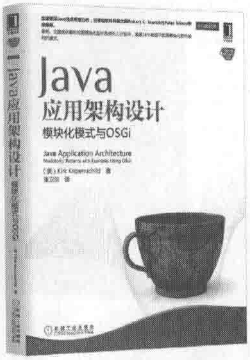

Java
核心技术 卷 I
基础知识


Java
核心技术 卷II
高级特性

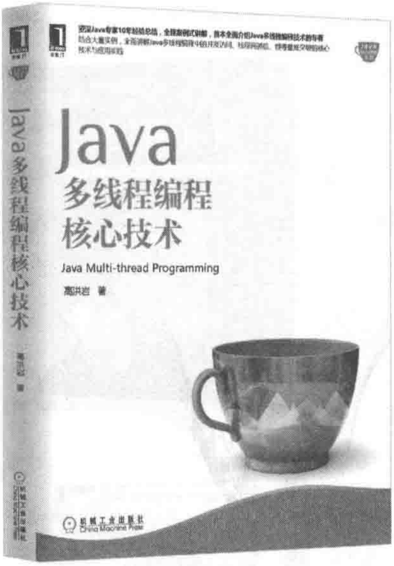

Java
虚拟机规范
（Java SE 7版）


Java
RESTful Web Service实战


Java EE 7
权威指南 卷 I


Java
应用架构设计
模块化模式与OSGi

推荐阅读

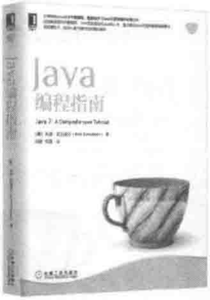


Java多线程编程核心技术

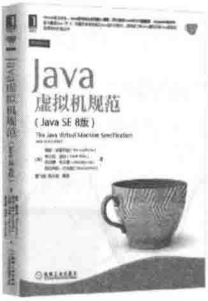
作者：高洪岩 ISBN：978-7-111-50206-7 定价：69.00元

资深Java专家10年经验总结，全程案例式讲解，首本全面介绍Java多线程编程技术的专著。
结合大量实例，全面讲解Java多线程编程中的并发访问、线程间通信、
锁等最难突破的核心技术与应用实践。

作者：沙伦 · 比奥卡 · 扎卡沃 等
ISBN：978-7-111-50392-7
定价：79.00元



作者：布迪 · 克尼亚万
ISBN：978-7-111-50381-1
定价：99.00元



作者：蒂姆 · 林霍尔姆 等
ISBN：978-7-111-50159-6
定价：79.00元

王 栋 雪球CTO

去年雪球花费了很大精力对内部系统进行了微服务化改造，我们选择了SpringBoot作为微服务化的基础，当时关于SpringBoot相关的中文资料还很少。最近听说福强老师写了一本关于SpringBoot的书，当看到书的目录后第一感觉就是如果我们当初有这本书在手绝对可以少走很多弯路。本书介绍了很多我们不曾研究透彻的用法和原理。更让人惊喜的是本书还包含关于微服务的原理和最佳实践的章节，让人耳目一新，收获颇丰。强烈推荐给每一个对SpringBoot和微服务感兴趣的人。

杜 江 21CTO社区创始人

福强的第一本书曾给我耳目一新的感觉，现在他的新作又一次带给我惊喜。和其他书籍相比，本书既在道上深入一层，又在术的层面，对SpringBoot庖丁解牛，对现在流行的微服务讲解得淋漓尽致，是为精品。若我想在SpringBoot上精进一层，则必读此书，然后请王老师喝到扶墙为止。

曹祖鹏 千米网首席架构师

本书从宏观上介绍了微服务的架构理论，又从技术实现上解析了SpringBoot的各种微妙之处。这本书理论和实践兼备，来得非常及时。

上架指导：计算机/程序设计/Java

ISBN 978-7-111-53664-2

定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn